

USENIX Association

**Proceedings of the
2022 USENIX Annual Technical Conference**

**July 11–13, 2022
Carlsbad, CA, USA**

Conference Organizers

Program Co-Chairs

Jiri Schindler, *Tranquil Data*

Noa Zilberman, *University of Oxford*

Program Committee

Reto Achermann, *University of British Columbia*

Gustavo Alonso, *ETH Zurich*

Raja Appuswamy, *EURECOM*

Anys Bacha, *University of Michigan*

Saurabh Bagchi, *Purdue University*

Yungang Bao, *Institute of Computing Technology, Chinese Academy of Sciences*

Antonio Barbalace, *University of Edinburgh*

Yaniv Ben Itzhak, *VMware Research*

Annette Bieniusa, *TU Kaiserslautern*

Roberto Bifulco, *NEC Laboratories Europe*

Laurent Bindschaedler, *Massachusetts Institute of Technology*

William Bolosky, *Microsoft Research*

James Bottomley, *IBM Research*

Nathan Bronson, *Rockset*

Mihai Budiu, *VMware Research*

Somali Chaterji, *Purdue University*

Lydia Chen, *Delft University of Technology*

Young-ri Choi, *UNIST (Ulsan National Institute of Science and Technology)*

David Cock, *ETH Zurich*

Dilma Da Silva, *Texas A&M University*

Angela Demke Brown, *University of Toronto*

Fred Douglass, *Peraton Labs*

Abhinav Duggal, *Dell EMC*

Pascal Felber, *University of Neuchatel*

Pedro Fonseca, *Purdue University*

Wei Gao, *University of Pittsburgh*

Eran Gilad, *Yahoo Research*

Yotam Harchol, *DFINITY Foundation*

Tim Harris, *Microsoft*

Niranjan Hasabnis, *Intel Labs*

David Hay, *Hebrew University*

Michio Honda, *University of Edinburgh*

Jon Howell, *VMware*

Yu Hua, *Huazhong University of Science and Technology*

Joo-young Hwang, *Samsung Electronics*

Rebecca Isaacs, *Twitter*

Zsolt Istvan, *TU Darmstadt*

Anand Iyer, *Microsoft Research*

Bill Jannen, *Williams College*

Theo Jepsen, *Stanford University*

Anuj Kalia, *Microsoft*

Michael Kozuch, *Intel Labs*

John Kubiatowicz, *University of California, Berkeley*

Youngjin Kwon, *Korea Advanced Institute of Science and Technology (KAIST)*

Sándor Laki, *ELTE Eötvös Loránd University*

Shir Landau Feibish, *The Open University of Israel*

Alberto Lerner, *University of Fribourg*

Youyou Lu, *Tsinghua University*

Xiaosong Ma, *Qatar Computing Research Institute*

Ilias Marinos, *Microsoft Research*

A. Theodore Marketos, *University of Cambridge*

Ali Mashtizadeh, *University of Waterloo*

Michael Mesnier, *Intel*

Ethan Miller, *University of California, Santa Cruz / Pure Storage*

Changwoo Min, *Virginia Tech*

Subrata Mitra, *Adobe Research*

Jayashree Mohan, *Microsoft Research India*

Sue Moon, *Korea Advanced Institute of Science and Technology (KAIST)*

Kiran-Kumar Muniswamy-Reddy, *Amazon*

Onur Mutlu, *ETH Zurich*

Khanh Nguyen, *Texas A&M University*

Ruslan Nikolaev, *The Pennsylvania State University*

Shadi Noghabi, *Microsoft Research*

Fernando Pedone, *Università della Svizzera italiana*

Adrian Perrig, *ETH Zurich*

Babu Pillai, *Intel Labs*

Thanumalayan Pillai, *Google*

Fernando Ramos, *Universidade de Lisboa*

Kaveh Razavi, *ETH Zurich*

Elissa M. Redmiles, *Max Planck Institute for Software Systems*

Larry Rudolph, *Two Sigma Investments, LP*

Russell Sears, *Apple*

Mark Silberstein, *Technion—Israel Institute of Technology*

Georgios Smaragdakis, *Delft University of Technology*

Keith A. Smith, *MongoDB*

Ripduman Sohan, *Xilinx*

Patrick Stuedi, *Meta*

Nik Sultana, *Illinois Institute of Technology*

Vasily Tarasov, *IBM Research - Almaden*

Alain Tchana, *ENS Lyon, France*

Jens Teubner, *TU Dortmund*

Eno Thereska, *Amazon*

Daniel Thomas, *University of Strathclyde*

Theodore Ts'o, *Google*

Shay Vargafik, *VMware Research*

Nandita Vijaykumar, *University of Toronto*

Haris Volos, *University of Cyprus*

Keval Vora, *Simon Fraser University*

Han Wang, *Intel*

Ric Wheeler, *Facebook*

Avani Wildani, *Emory University*

Dan Williams, *Virginia Tech*

Youjip Won, *Korea Advanced Institute of Science and Technology (KAIST)*

Eiko Yoneki, *University of Cambridge*

Yibo Zhu, *ByteDance*

© 2022 by The USENIX Association

All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-1-939133-29-8

External Reviewers

Amit Levy
Aravind Machiry
Calin Iorgulescu

Chia-Che Tsai
Jasmina Malicevic
Minjia Zhang

Reza Yazdani Aminabadi
Roberto Palmieri
Valerio Schiavoni

Message from the USENIX ATC '22 Program Co-Chairs

Introduction

Welcome to the 2022 USENIX Annual Technical Conference (ATC). We are excited that, after two years of being virtual due to the COVID-19 pandemic, this year’s conference is held again in person. Because of the ongoing special circumstances, USENIX has adopted a hybrid model with some attendees and presenters connecting remotely.

Similar to last year, ATC 2022 is co-located with OSDI. Our 2021 predecessors have already written extensively about the opportunities and challenges of running two systems conferences at the same time (https://www.usenix.org/sites/default/files/atc21_message.pdf). This year brings the new challenge of running the two co-located events in a hybrid model. We very much look forward to meeting everyone in the systems community whether they attend ATC, OSDI, or both.

The rest of this document provides some insights into the submission and selection process that culminated in 64 accepted works that will be presented at the conference.

Submissions process

We have solicited full length and short papers presenting new and original computer systems work. We adopted a double-blind review process to minimize bias. To further the USENIX mission of bringing together researchers in academia and systems practitioners, we have designated a special Operational Systems Track (OST) category to solicit submissions describing the experiences from deployed systems at “production” scale with real-world data. OST submissions received the same rigorous review process but with different criteria. The submission’s novelty bar was lower, and system and organization names did not have to be blinded. Switching submission tracks after the deadline was forbidden.

Authors were requested to provide additional information with their submission. First, we asked whether the paper was a re-submission from prior ATC or some other conference. 65% of the papers were marked as first-time submissions and 45% of the accepted papers were first time submissions. In case of a resubmission, authors provided a description of what changes they made since the previous submission. The reviewers and the program committee (PC) had access to this information, but they did not know the venue where the paper was submitted or specific review comments (unless provided by the authors). Prior submission information had no bearing on assigning reviewers.

We also asked the authors to indicate whether they would make an artifact available. 70% of submissions indicated they would, if accepted. With all else being equal, the PC viewed more favorably submissions that would share an artifact over those that did not. As researchers, we need to ensure reproducibility of published works. As members of the USENIX community, we want to provide free and open access to data. The artifact evaluation process, which we instituted this year together with OSDI, provides this assurance.

We received 394 submissions, of which 21 (5%) were in OST and 23 (6%) were short papers. This was about 15% more submissions than in the previous two years. We rejected 5 submissions without a review due to violating one or more directives stated in the call for papers (CFP). The most popular submission topics were Distributed System (26%), Storage (24%), Machine Learning (21%), Operating Systems (15%), Networking (14%), Databases (13%) and Security (13%).

In the end, the PC accepted 64 submissions for an overall 16.5% acceptance rate. Acceptance was based on the quality of the submissions, while in-person conference constraints had no bearing on our decisions. Of the 64 accepted submissions, 7 (33% acceptance rate) were in Operational Systems Track and 2 (9% acceptance rate) were short papers.

Program Committee

We have assembled a program committee with many goals in mind: good coverage across diverse computer systems topics, balance between academia and industry, a mix of veterans of prior ATC PCs with individuals in early stages of their professional careers, geographic diversity, and adherence to the USENIX diversity and inclusion principles.

The assembled PC had 97 members from 15 countries, 52% from North America, 37% from EMEA and 10% from APAC. 60% of the PC were from academia and 40% from industry, though some PC members from academia were also affiliated with industry. 36% of the reviewers were early career researchers. Women were 64% more likely to decline an invitation to join the PC, which we find to be an alarming indication.

The main areas of expertise of PC members were Storage (22%), Distributed Systems (20%), Operating Systems (14%), Security (13%), Networking (12%) and Databases (9%). This was a good match to the submissions topics, given the PC was assembled in advance. As only 9% of PC members indicated that Machine Learning is their main expertise, a mismatch with 21% of submissions, we expanded the PC post submission deadline with more machine learning experts and recruited the help of a few expert external reviewers.

Reviewing Process

We proceeded with two double-blind review rounds with the authors’ response after round 2 and before the PC meeting. We sent early rejection notifications to 58% of papers 10 weeks after the initial submission to allow authors a quick turnaround on their resubmission. In the first round, we assigned 3 reviewers per paper, in the vast majority of cases, complementing the expertise with external reviewers as necessary. In the second round, we assigned at least two additional reviewers to the 162 submissions not rejected earlier.

After the authors’ response and an online discussion among the reviewers (with some papers receiving over 20 comments), we pre-accepted 48 papers. We identified additional 39 papers for discussion at the face-to-face (virtual) PC meeting, of which 42% of papers were accepted, and PC members had the opportunity to “revive” papers. Despite having PC members spanning a geographic area of 13 time zones, we conducted the virtual meeting “live”. While the day (and night) was long, with the usual logistical challenges of handling conflicts virtually in break-out rooms, we found that the ability to discuss and calibrate our acceptance criteria during the PC meeting was very important and proved very useful.

Artifact Evaluation Process

For the first time this year, ATC adopted an artifact evaluation process. The process ran jointly with OSDI, led by Anuj Kalia, Neeraja J. Yadwadkar, and Chengyu Zhang. The artifact evaluation committee chairs assembled a committee consisting of 118 members.

The authors of all accepted papers were invited to submit an artifact for an evaluation. 52 out of 64 papers (81%) had done so. 88% of artifacts received an “Available” badge, 76% received a “Functional” badge, and 61% received a “Reproduced” badge. 51% of papers received all three badges (some artifacts were reproduced, but are not available). Only one artifact received no badge.

Additional Observations

Strong papers easily stood out; 38% of the accepted papers received only positive reviews, and an additional 44% had only a single weak-reject initial recommendation. This is also why so many papers were accepted prior to the PC discussion.

The Operational System Track (OST) was intended only for operational systems, especially those deployed at scale. In particular, there was an interest in the experience using these operational systems. Some authors mistook a working prototype for an operational system. While all submissions to ATC are expected to describe working systems, a prototype implementation to gather experimental results is not the same. OST submissions required describing the experience of using the system.

Anonymization rules were not always followed. Only one paper was rejected immediately during post-submission checks for blinding authors names and affiliations. However, several papers were rejected following the first round of reviews, as authors had a technical report using a similar title or a similar system name. This, in turn, led to unblinding the papers to reviewers and violated the submission rules. Anonymization rules, especially when applied to technical reports, vary from year to year and between conferences. Authors should be extra vigilant when submitting blinded manuscripts.

Acknowledgements

More than 200 people have contributed to the organization of the USENIX ATC '22, most of them in a voluntary capacity. We would like to thank each and every one of them. We are tremendously grateful to the program committee members for a job extremely well done, and for their personal sacrifices. We thank the Artifact Evaluation committee and the Artifact Evaluation Committee Chairs for their work and contribution, which improves our community and enables future research. Last, we thank the USENIX organization, the USENIX ATC steering committee and OSDI '22 co-chairs. The amount of work and preparation that goes into organizing a conference is immense, and we were astounded by the help and support provided by everyone involved.

USENIX ATC '22 Program Co-Chairs
Noa Zilberman, *University of Oxford*
Jiri Schindler, *Tranquil Data*

2022 USENIX Annual Technical Conference

July 11–13, 2022

Monday, July 11

Storage 1

ZNSwap: un-Block your Swap 1
Shai Bergman, *Technion*; Niklas Cassel and Matias Bjørling, *Western Digital*; Mark Silberstein, *Technion*

Building a High-performance Fine-grained Deduplication Framework for Backup Storage with High Deduplication Ratio 19
Xiangyu Zou and Wen Xia, *Harbin Institute of Technology, Shenzhen*; Philip Shilane, *Dell Technologies*; Haijun Zhang and Xuan Wang, *Harbin Institute of Technology, Shenzhen*

Secure and Lightweight Deduplicated Storage via Shielded Deduplication-Before-Encryption 37
Zuoru Yang, *The Chinese University of Hong Kong*; Jingwei Li, *University of Electronic Science and Technology of China*; Patrick P. C. Lee, *The Chinese University of Hong Kong*

Containers

RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing 53
Zijun Li, *Department of Computer Science and Engineering, Shanghai Jiao Tong University and Alibaba Group*; Jiagan Cheng, and Quan Chen, *Department of Computer Science and Engineering, Shanghai Jiao Tong University*; Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, and Weidong Han, *Alibaba Group*; Minyi Guo, *Department of Computer Science and Engineering, Shanghai Jiao Tong University*

Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing 69
Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, and Chuhao Xu, *Shanghai Jiao Tong University*; Deze Zeng, *School of Computer Science, China University of Geosciences*; Zhuo Song, Tao Ma, and Yong Yang, *Alibaba Cloud*; Chao Li and Minyi Guo, *Department of Computer Science and Engineering, Shanghai Jiao Tong University*

RRC: Responsive Replicated Containers 85
Diyu Zhou, *UCLA and EPFL*; Yuval Tamir, *UCLA*

Distributed Systems 1

uKharon: A Membership Service for Microsecond Applications 101
Rachid Guerraoui and Antoine Murat, *École Polytechnique Fédérale de Lausanne (EPFL)*; Javier Picorel, *Huawei Technologies*; Athanasios Xygkis, *EPFL*; Huabing Yan and Pengfei Zuo, *Huawei Technologies*

KRCORE: A Microsecond-scale RDMA Control Plane for Elastic Computing 121
Xingda Wei, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, and Shanghai AI Laboratory*; Fangming Lu, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University*; Rong Chen, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University, and Shanghai AI Laboratory*; Haibo Chen, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University*

Zero-Change Object Transmission for Distributed Big Data Analytics 137
Mingyu Wu, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University and Shanghai AI Laboratory*; Shuaiwei Wang, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University*; Haibo Chen and Binyu Zang, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University and Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*

Sift: Using Refinement-guided Automation to Verify Complex Distributed Systems 151
Haojun Ma, Hammad Ahmad, Aman Goel, Eli Goldweber, Jean-Baptiste Jeannin, Manos Kapritsos, and Baris Kasikci, *University of Michigan*

Machine Learning 1

Faith: An Efficient Framework for Transformer Verification on GPUs 167
Boyuan Feng, Tianqi Tang, Yuke Wang, Zhaodong Chen, Zheng Wang, Shu Yang, Yuan Xie, and Yufei Ding, *University of California, Santa Barbara*

DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs . 183
Weihao Cui, Han Zhao, Quan Chen, Hao Wei, and Zirui Li, *Shanghai Jiao Tong University*; Deze Zeng, *China University of Geosciences*; Chao Li and Minyi Guo, *Shanghai Jiao Tong University*

Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing 199
Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh, *KAIST*

PilotFish: Harvesting Free Cycles of Cloud Gaming with Deep Learning Training 217
Wei Zhang and Binghao Chen, *Shanghai Jiao Tong University*; Zhenhua Han, *Microsoft Research Asia*; Quan Chen, *Shanghai Jiao Tong University*; Peng Cheng, Fan Yang, Ran Shu, and Yuqing Yang, *Microsoft Research*; Minyi Guo, *Shanghai Jiao Tong University*

Operating Systems 1

Privbox: Faster System Calls Through Sandboxed Privileged Execution 233
Dmitry Kuznetsov and Adam Morrison, *Tel Aviv University*

BBQ: A Block-based Bounded Queue for Exchanging Data and Profiling 249
Jiawei Wang, *Huawei Dresden Research Center, Huawei OS Kernel Lab and Technische Universität Dresden*; Diogo Behrens, Ming Fu, Lilith Oberhauser, Jonas Oberhauser, and Jitang Lei, *Huawei Dresden Research Center, Huawei OS Kernel Lab*; Geng Chen, *Huawei OS Kernel Lab*; Hermann Härtig, *Technische Universität Dresden*; Haibo Chen, *Huawei OS Kernel Lab and Shanghai Jiao Tong University*

Disaggregated Systems

Sibylla: To Retry or Not To Retry on Deep Learning Job Failure 263
Taeyoon Kim, Suyeon Jeong, Jongseop Lee, Soobee Lee, and Myeongjae Jeon, *UNIST*

Speculative Recovery: Cheap, Highly Available Fault Tolerance with Disaggregated Storage 271
Nanqinqin Li, Anja Kalaba, Michael J. Freedman, Wyatt Lloyd, and Amit Levy, *Princeton University*

Direct Access, High-Performance Memory Disaggregation with DirectCXL 287
Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung, *Computer Architecture and Memory Systems Laboratory, Korea Advanced Institute of Science and Technology (KAIST)*

Networking 1

Not that Simple: Email Delivery in the 21st Century 295
Florian Holzbauer, *SBA Research*; Johanna Ullrich, *University of Vienna*; Martina Lindorfer, *TU Wien*; Tobias Fiebig, *Max-Planck-Institut für Informatik*

AddrMiner: A Comprehensive Global Active IPv6 Address Discovery System 309
Guanglei Song, Jiahai Yang, Lin He, Zhiliang Wang, *Institute for Network Sciences and Cyberspace, BNRist, Tsinghua University and Quan Cheng Laboratory, Jinan, Shandong, China*; Guo Li and Chenxin Duan, *Institute for Network Sciences and Cyberspace, BNRist, Tsinghua University*; Yaozhong Liu, *Tsinghua University*; Zhongxiang Sun, *School of Computer and Information Technology, Beijing Jiaotong University*

Co-opting Linux Processes for High-Performance Network Simulation 327
Rob Jansen, *U.S. Naval Research Laboratory*; Jim Newsome, *Tor Project*; Ryan Wails, *Georgetown University, U.S. Naval Research Laboratory*

Finding Bugs

KSG: Augmenting Kernel Fuzzing with System Call Specification Generation 351
Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang, *Tsinghua University*

DLOS: Effective Static Detection of Deadlocks in OS Kernels 367
Jia-Ju Bai, Tuo Li, and Shi-Min Hu, *Tsinghua University*

Modulo: Finding Convergence Failure Bugs in Distributed Systems with Divergence Resync Models 383
Beom Heyn Kim, *Samsung Research and University of Toronto*; Taesoo Kim, *Samsung Research and Georgia Institute of Technology*; David Lie, *University of Toronto*

Tuesday, July 12

Security

SoftTRR: Protect Page Tables Against Rowhammer Attacks Using Software-Only Target Row Refresh **399**

Zhi Zhang, *CSIRO's Data61, Australia*; Yueqiang Cheng, *NIO Security Research*; Minghua Wang, *Baidu Security*; Wei He and Wenhao Wang, *State Key Laboratory of Information Security, Institute of Information Engineering, CAS, and University of Chinese Academy of Sciences*; Surya Nepal, *CSIRO's Data61, Australia*; Yansong Gao, *Nanjing University of Science and Technology, China*; Kang Li, *Baidu Security*; Zhe Wang and Chenggang Wu, *State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS, and University of Chinese Academy of Sciences*

Hardening Hypervisors with Ombro **415**

Ethan Johnson, Colin Pronovost, and John Criswell, *Department of Computer Science, University of Rochester*

HyperEnclave: An Open and Cross-platform Trusted Execution Environment **437**

Yuekai Jia, *Tsinghua University*; Shuang Liu, *Ant Group*; Wenhao Wang, *SKLOIS, Institute of Information Engineering, CAS, and School of Cyber Security, University of Chinese Academy of Sciences*; Yu Chen, *Tsinghua University*; Zhengde Zhai, Shoumeng Yan, and Zhengyu He, *Ant Group*

PRIDWEN: Universally Hardening SGX Programs via Load-Time Synthesis **455**

Fan Sang, *Georgia Institute of Technology*; Ming-Wei Shih, *Microsoft*; Sangho Lee, *Microsoft Research*; Xiaokuan Zhang, *Georgia Institute of Technology*; Michael Steiner and Mona Vij, *Intel Labs*; Taesoo Kim, *Georgia Institute of Technology*

Machine Learning 2

Tetris: Memory-efficient Serverless Inference through Tensor Sharing **473**

Jie Li, Laiping Zhao, and Yanan Yang, *College of Intelligence & Computing (CIC), Tianjin University, and Tianjin Key Lab of Advanced Networking (TANKLAB)*; Kunlin Zhan, *58.com*; Keqiu Li, *College of Intelligence & Computing (CIC), Tianjin University, and Tianjin Key Lab of Advanced Networking (TANKLAB)*

PetS: A Unified Framework for Parameter-Efficient Transformers Serving **489**

Zhe Zhou, *Peking University*; Xuechao Wei, *Peking University and Alibaba Group*; Jiejing Zhang, *Alibaba Group*; Guangyu Sun, *Peking University*

Campo: Cost-Aware Performance Optimization for Mixed-Precision Neural Network Training **505**

Xin He, *CSEE, Hunan University & Xidian University*; Jianhua Sun and Hao Chen, *CSEE, Hunan University*; Dong Li, *University of California, Merced*

Primo: Practical Learning-Augmented Systems with Interpretable Models **519**

Qinghao Hu, *Nanyang Technological University and S-Lab, NTU*; Harsha Nori, *Microsoft*; Peng Sun, *SenseTime*; Yonggang Wen and Tianwei Zhang, *Nanyang Technological University*

Distributed Systems 2

Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems **539**

Rong Gu, Han Yin, Weichang Zhong, Chunfeng Yuan, and Yihua Huang, *State Key Laboratory for Novel Software Technology, Nanjing University*

DepFast: Orchestrating Code of Quorum Systems **557**

Xuhao Luo, *University of Illinois at Urbana-Champaign*; Weihai Shen and Shuai Mu, *Stony Brook University*; Tianyin Xu, *University of Illinois at Urbana-Champaign*

High Throughput Replication with Integrated Membership Management **575**

Pedro Fouto, Nuno Preguiça, and João Leitão, *NOVA LINCS & NOVA University Lisbon*

Operating Systems 2

CBMM: Financial Advice for Kernel Memory Managers **593**

Mark Mansi, Bijan Tabatabai, and Michael M. Swift, *University of Wisconsin - Madison*

EPK: Scalable and Efficient Memory Protection Keys **609**

Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen, *Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China, Institute of Parallel and Distributed Systems (IPADS), SEIEE, Shanghai Jiao Tong University*

Memory Harvesting in Multi-GPU Systems with Hierarchical Unified Virtual Memory **625**

Sangjin Choi and Taeksoo Kim, *KAIST*; Jinwoo Jeong, *Ajou University*; Rachata Ausavarungnirun, *KMUTNB*; Myeongjae Jeon, *UNIST*; Youngjin Kwon, *KAIST*; Jeongseob Ahn, *Ajou University*

Deployed Systems 1

Zero Overhead Monitoring for Cloud-native Infrastructure using RDMA **639**

Zhe Wang, *Shanghai Jiao Tong University*; Teng Ma, *Alibaba Group*; Linghe Kong, *Shanghai Jiao Tong University*; Zhenzao Wen, Jingxuan Li, Zhuo Song, Yang Lu, Yong Yang, and Tao Ma, *Alibaba Group*; Guihai Chen, *Shanghai Jiao Tong University*; Wei Cao, *Alibaba Group*

CRISP: Critical Path Analysis of Large-Scale Microservice Architectures **655**

Zhizhou Zhang, *UC Santa Barbara*; Murali Krishna Ramanathan, Prithvi Raj, and Abhishek Parwal, *Uber Technologies Inc.*; Timothy Sherwood, *UC Santa Barbara*; Milind Chabbi, *Uber Technologies Inc.*

Whale: Efficient Giant Model Training over Heterogeneous GPUs **673**

Xianyan Jia, Le Jiang, Ang Wang, and Wencong Xiao, *Alibaba Group*; Ziji Shi, *Alibaba Group and National University of Singapore*; Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin, *Alibaba Group*

Machine Learning 3

Cachew: Machine Learning Input Data Processing as a Service **689**

Dan Graur, Damien Aymon, Dan Kluser, and Tanguy Albrici, *ETH Zurich*; Chandramohan A. Thekkath, *Google*; Ana Klimovic, *ETH Zurich*

CoVA: Exploiting Compressed-Domain Analysis to Accelerate Video Analytics **707**

Jinwoo Hwang, Minsu Kim, Daeun Kim, Seungho Nam, Yoosung Kim, and Dohee Kim, *KAIST*; Hardik Sharma, *Google*; Jongse Park, *KAIST*

SOTER: Guarding Black-box Inference for General Neural Networks at the Edge **723**

Tianxiang Shen, Ji Qi, Jianyu Jiang, Xian Wang, Siyuan Wen, Xusheng Chen, and Shixiong Zhao, *The University of Hong Kong*; Sen Wang and Li Chen, *Huawei Technologies*; Xiapu Luo, *The Hong Kong Polytechnic University*; Fengwei Zhang, *Southern University of Science and Technology (SUSTech)*; Heming Cui, *The University of Hong Kong*

Storage 2

IPLFS: Log-Structured File System without Garbage Collection **739**

Juwon Kim, Minsu Jang, Muhammad Danish Tehseen, Joontaek Oh, and YouJip Won, *KAIST*

Vigil-KV: Hardware-Software Co-Design to Integrate Strong Latency Determinism into Log-Structured Merge Key-Value Stores **755**

Miryeong Kwon, Seungjun Lee, and Hyunkyu Choi, *KAIST*; Jooyoung Hwang, *Samsung Electronics Co., Ltd.*; Myoungsoo Jung, *KAIST*

Pacman: An Efficient Compaction Approach for Log-Structured Key-Value Store on Persistent Memory **773**

Jing Wang, Youyou Lu, Qing Wang, and Minhui Xie, *Tsinghua University*; Keji Huang, *Huawei Technologies Co., Ltd*; Jiwu Shu, *Tsinghua University*

Networking 2

Towards Latency Awareness for Content Delivery Network Caching **789**

Gang Yan and Jian Li, *SUNY-Binghamton University*

Hashing Design in Modern Networks: Challenges and Mitigation Techniques **805**

Yunhong Xu, *Texas A&M University*; Keqiang He and Rui Wang, *Google*; Minlan Yu, *Harvard University and Google*; Nick Duffield, *Texas A&M University*; Hassan Wassel, Shidong Zhang, Leon Poutievski, Junlan Zhou, and Amin Vahdat, *Google*

Firebolt: Finding Bugs in Programmable Data Plane Generators **819**

Jiamin Cao, *Tsinghua University*; Yu Zhou and Chen Sun, *Alibaba Group*; Lin He, Zhaowei Xi, and Ying Liu, *Tsinghua University*

Wednesday, July 13

Compilers and PL

- Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster?** 835
David Lion, *University of Toronto and YScope Inc.*; Adrian Chiu and Michael Stumm, *University of Toronto*; Ding Yuan, *University of Toronto and YScope Inc.*
- Automatic Recovery of Fine-grained Compiler Artifacts at the Binary Level** 853
Yufei Du, *University of North Carolina at Chapel Hill*; Ryan Court and Kevin Snow, *Zeropoint Dynamics*; Fabian Monroe, *University of North Carolina at Chapel Hill*
- JITServer: Disaggregated Caching JIT Compiler for the JVM in the Cloud** 869
Alexey Khrabrov, *University of Toronto*; Marius Pirvu and Vijay Sundaesan, *IBM*; Eyal de Lara, *University of Toronto*
- Riker: Always-Correct and Fast Incremental Builds from Simple Specifications.** 885
Charlie Curtsinger, *Grinnell College*; Daniel W. Barowy, *Williams College*

Storage 3

- FlatFS: Flatten Hierarchical File System Namespace on Non-volatile Memories** 899
Miao Cai, *Key Laboratory of Water Big Data Technology of Ministry of Water Resources, Hohai University; School of Computer and Information, Hohai University; State Key Laboratory for Novel Software Technology, Nanjing University*; Junru Shen, *School of Computer and Information, Hohai University*; Bin Tang, *School of Computer and Information, Hohai University*; Hao Huang, *State Key Laboratory for Novel Software Technology, Nanjing University*; Baoliu Ye, *State Key Laboratory for Novel Software Technology, Nanjing University*; Key Laboratory of Water Big Data Technology of Ministry of Water Resources, Hohai University; School of Computer and Information, Hohai University
- StRAID: Stripe-threaded Architecture for Parity-based RAIDs with Ultra-fast SSDs** 915
Shucheng Wang, Qiang Cao, and Ziyi Lu, *Wuhan National Laboratory for Optoelectronics, HUST*; Hong Jiang, *Department of Computer Science and Engineering, UT Arlington*; Jie Yao, *School of Computer Science and Technology, HUST*; Yuanyuan Dong, *Alibaba Group*
- VINTER: Automatic Non-Volatile Memory Crash Consistency Testing for Full Systems** 933
Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa, *Karlsruhe Institute of Technology*

NICs

- AINiCo: SmartNIC-accelerated Contention-aware Request Scheduling for Transaction Processing.** 951
Junru Li, Youyou Lu, Qing Wang, Jiazhen Lin, Zhe Yang, and Jiwu Shu, *Beijing National Research Center for Information Science and Technology (BNRist)*
- FpgaNIC: An FPGA-based Versatile 100Gb SmartNIC for GPUs** 967
Zeke Wang, Hongjing Huang, Jie Zhang, *Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, China*; Fei Wu, *Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, China, and Shanghai Institute for Advanced Study of Zhejiang University, China*; Gustavo Alonso, *ETH Zurich*
- Faster Software Packet Processing on FPGA NICs with eBPF Program Warping.** 987
Marco Bonola, *CNIT/Axbryd*; Giacomo Belocchi, Angelo Tulumello, and Marco Spaziani Brunella, *Axbryd/University of Rome Tor Vergata*; Giuseppe Siracusano, *NEC Laboratories Europe*; Giuseppe Bianchi, *University of Rome Tor Vergata*; Roberto Bifulco, *NEC Laboratories Europe*

Deployed Systems 2

- NVMe SSD Failures in the Field: the Fail-Stop and the Fail-Slow** 1005
Ruiming Lu, *Shanghai Jiao Tong University*; Erci Xu, *PDL*; Yiming Zhang, *Xiamen University*; Zhaosheng Zhu, Mengtian Wang, and Zongpeng Zhu, *Alibaba Inc.*; Guangtao Xue, *Shanghai Jiao Tong University*; Minglu Li, *Shanghai Jiao Tong University & Zhejiang Normal University*; Jiasheng Wu, *Alibaba Inc.*
- CacheSack: Admission Optimization for Google Datacenter Flash Caches.** 1021
Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister, *Google*
- Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service** 1037
Mostafa Elhemali, Niall Gallagher, Nicholas Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somu Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sothikul, Doug Terry, Akshat Vig, *Amazon Web Services*

ZNSwap: un-Block your Swap

Shai Bergman
Technion

Niklas Cassel
Western Digital

Matias Bjørling
Western Digital

Mark Silberstein
Technion

Abstract

We introduce *ZNSwap*, a novel swap subsystem optimized for the recent Zoned Namespace (ZNS) SSDs. ZNSwap leverages ZNS’s explicit control over data management on the drive and introduces a space-efficient host-side Garbage Collector (GC) for swap storage co-designed with the OS swap logic. ZNSwap enables cross-layer optimizations, such as direct access to the in-kernel swap usage statistics by the GC to enable fine-grain swap storage management, and correct accounting of the GC bandwidth usage in the OS resource isolation mechanisms to improve performance isolation in multi-tenant environments. We evaluate ZNSwap using standard Linux swap benchmarks and two production key-value stores. ZNSwap shows significant performance benefits over the Linux swap on traditional SSDs, such as stable throughput for different memory access patterns, and $10\times$ lower 99th percentile latency and $5\times$ higher throughput for `memcached` key-value store under realistic usage scenarios.

1 Introduction

Swap is regaining interest from the academia, industry, and kernel communities [2, 3, 12–14, 42, 43, 53, 54] as SSDs are getting faster with both low-latency NAND and high-speed PCIe interfaces [5, 11, 51]. Swap on SSDs is no longer viewed as a last-resort memory-overflow mechanism, but as a crucial system component essential for effective memory reclamation and high system efficiency [3, 14, 18].

Unfortunately, the broader deployment of SSDs as swap devices is overshadowed by several notable performance issues. One of the key limitations is the system performance degradation as the SSD utilization increases. For example, [Figure 1](#) shows a drastic swap bandwidth drop as the device usage grows beyond 20%, forcing low space utilization to maintain high performance. In § 3 we thoroughly analyze this and other performance issues with swap on SSDs, such as bandwidth variations for different memory access patterns, and poor isolation in a multi-tenant setting.

These performance anomalies have no simple solution. They stem from the inherent mismatch between the easy-to-use block-interface abstraction and the intrinsic flash media

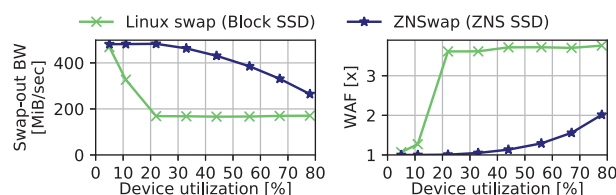


Figure 1: Swap-out bandwidth of random memory accesses (a common swap access pattern [43, 55]), with default Linux swap on Block SSD and ZNSwap on ZNS SSD. The two 1TB SSDs share the same hardware platform and media. WAF—Write Amplification Factor.

properties. In particular, this interface deliberately conceals the absence of in-place updates to flash-based media. Under the hood, updates are written out-of-place to a specifically allocated set of flash blocks (i.e., erase-block). To this end, SSD controllers implement a Flash Translation Layer (FTL), which translates the host-side random writes into sequential writes required by the media, and maintains logic-to-physical mapping for each block. It further entails a device-side Garbage Collection (GC) process to free up erase blocks and reclaim capacity for new writes.

More crucially, this interface decouples the media management from the host-side software stack, so neither the software using the SSD nor the SSD’s management logic have any visibility into each other activities. In the context of swap, this decoupling hinders the OS’s ability to optimize data placement on the device, and the device’s ability to leverage unique characteristics of the swap mechanisms and its usage of the device. For example, the performance degradation observed in [Figure 1](#) is caused by *Write Amplification (WA)*, i.e., the extra data movements performed by the GC. Notably, as we show in § 3, the Write Amplification Factor (WAF) ([Figure 1](#), right) could have been reduced if only the GC were aware of the OS-managed validity status of the stored blocks.

Zoned Storage interface for SSDs (ZNS) [4] aims to reestablish the host’s control over key aspects of the storage device management [25]. ZNS opens unique opportunities for cross-layer optimizations that allow novel storage-application co-design simultaneously tailored to the properties of the storage

media and its use by applications [25].

At a high level, ZNS introduces the concepts of zones. Zones disallow in-place updates and require their blocks to be written sequentially. To reclaim the space in a zone it needs to be reset, and new writes can be issued. One important benefit of the ZNS interface over prior attempts to expose flash media control to applications (i.e., raw flash or open-channel SSD [26]), is that it enables host-side storage control without having to deal with low-level media management such as wear-leveling or error correction.

In this work, we introduce **ZNSwap**, a novel swap subsystem for Linux that explores the advantages of the synergy between the SSD management and the OS swap logic, leveraging the ZNS interface to overcome the swap performance issues with block-interface SSDs. While prior works observed that the direct application control over SSDs can be beneficial in the context of file-systems and key-value stores [25, 26, 30, 59], ZNSwap is the first to leverage such control for the OS swap on SSDs.

ZNSwap provides a novel, space-efficient host-side mechanism for SSD space reclamation we call *ZNS Garbage Collector* (ZNGC). Unlike the device-side GC of traditional SSDs, ZNGC is tightly integrated with the OS and has direct access to OS data structures which it uses to optimize its operation.

ZNGC design poses a conceptual challenge, however. The space reclamation process naturally involves the migration of logical blocks on the device, without coordinating the block location changes with the applications that own the stored data. This is not a problem for an SSD-side GC because the user-visible Logical Block Addresses (LBA) remain intact. However, applying this solution to the host-side ZNGC would incur unacceptable space overheads in the host, requiring to maintain reverse mapping for every 4KiB block in TB-scale devices. ZNSwap avoids these overheads in the host by storing the reverse mapping information into the logical block metadata being written alongside the swapped-out page contents. The mapping is guaranteed to be correct during the page lifetime.

More specifically, ZNSwap brings the following benefits:

Fine-grain space management. ZNSwap obviates the need for TRIM commands, achieving higher performance and better space utilization. The OS uses TRIMs to hint to a Block SSD to deallocate specific LBAs, reducing the load on the SSD-side GC. Unfortunately, the use of TRIMs have been mostly disabled in the OS swap for their large overheads [35, 39, 50, 54], at the expense of significant bandwidth drop due to the artificial space bloat (§ 3.1.1). In ZNSwap, the ZNGC leverages the direct access to the OS internal page validity structures, without the costly overheads associated with TRIMs.

Dynamic ZNGC optimization. ZNSwap dynamically adjusts the number of swapped-in pages that are also stored in the swap device, improving the performance for read-mostly and mixed read-write workloads. The OS keeps a copy of unmodified swapped-in memory pages in the swap device

to avoid the swap-out penalty for those pages. The amount of disk space such pages may occupy is statically capped by the OS (50% in Linux, non-configurable). However, this static threshold does not fit all workloads: lower values degrade read-mostly workloads, whereas higher values affect mixed read-write workloads (§ 3.1.2). ZNSwap monitors the WAF and decreases the storage occupancy when necessary by reclaiming the SSD space from swapped-in pages.

Flexible data placement and space reclaim policies. ZNSwap allows easy customization of the disk space management policies to tailor the GC logic to the swap requirements of a specific system. For example, a policy may force co-location of data with similar lifetimes onto the same zone, which was shown to be useful before [28, 34, 44, 56], or achieve better performance isolation by dedicating a separate zone to handle swap from a specific tenant.

Accurate multi-tenancy accounting. As the ZNGC runs on the host, ZNSwap integrates with the cgroup accounting mechanisms to explicitly attribute GC overheads to different tenants, thus improving performance isolation between them.

To summarize, our main contributions are as follows:

- Thorough analysis of traditional Block SSDs' drawbacks when used as swap devices.
- A mechanism to enable ZNS SSDs to serve for swap, without resource-expensive redirection mechanisms in the host, by leveraging logical block metadata for efficient reverse-mapping.
- Custom swap-aware SSD storage management policies which reduce WA, improve performance, and achieve better isolation in multi-tenant environments.
- Extensive evaluation on standard benchmarks and real applications, demonstrating ZNSwap's performance gains, e.g., up to 10× lower 99th percentile latency and 5× higher throughput for `memcached`, with 2.5× lower WAF when compared to traditional swap on Block SSD.

2 Background

OS swap. When a system encounters memory pressure, it selects victim memory pages for eviction to a *swap device*. The OS unmaps the page chosen for eviction from the page-tables and *swaps-out* the page, writing it to the swap device.

Linux divides the space on a swap device into memory-page-sized blocks called *swap-slots*. The OS allocates a new slot for each page being swapped-out. When a page is swapped-in and the utilized swap device capacity is below 50%, Linux keeps the copy of the page both in memory and in the swap. Such pages belong to the *swap-cache*. The OS evicts swap-cache pages without writing them back to the swap. The swap-slot is freed upon the first write to a swap-cache page, and the page is removed from the swap-cache.

Block SSD space management. The SSD's FTL maps Logical Block Addresses (LBAs) to the physical data locations

within erase-blocks on the device. An update to a logical block is implemented by writing the new data to a separate erase-block, and then remapping the host-side LBA to the new block, followed by invalidating the old one. To free space for new writes, a *Garbage Collector* (GC), executed by the SSD controller, reclaims the invalidated blocks and consolidates the still-valid blocks from multiple erase-blocks to a new erase-block, and then erases the freed erase-blocks. This operation requires *over-provisioning* (OP) of the flash media in the drive in order to reduce the number of copies during the GC operation.

The device-side GC competes for bandwidth with the user I/O. The relative increase in the amount of data written due to GC vs. the external writes is called a *Write Amplification Factor* (WAF). The smaller the OP, the higher the WAF and the lower the user-visible performance of the device [34].

Zoned Namespace (ZNS) is a new storage interface for SSDs [25]. A ZNS SSD is organized as a set of logically-addressable *zones*. Each zone is physically aligned to an SSD's erase-block size. Reads inside a zone can be random, but writes must be *sequential*. Writing to a zone can be done via the common write command or through the *Zone Append* command. The latter works by the host specifying the zone, and the SSD returning the specific write location upon completion, which allows multiple in-flight requests to the same zone [24] (unlike the write command).

Each zone may be either `Empty` (initial state), `Open` (after the first write) or `Full` (no longer writable). The SSD maintains a write pointer to the next logical block for each `Open` zone. To rewrite a zone, it must be *reset*, which transitions it into an `Empty` state. There is a hardware limit on the number of simultaneously `Open` zones.

3 Motivation

Swap performance is important for data centers. The proliferation of fast flash-based storage revitalized the use of swap as a way to maximize memory utilization and reduce costs. Today, swapping does not serve for sustaining severe memory pressure alone. Rather, swap acts as a memory extension during moderate loads, e.g., to optimize the in-memory balance between file-backed and anonymous memory pages [3].

Thus, the swap performance is becoming increasingly important. Recent works propose to accelerate the swap with dedicated hardware [42]. Linux kernel added optimizations to its memory reclamation mechanism [13]. Alibaba Cloud added a per-cgroup background reclaim mechanism [12] to improve multi-tenancy support in data centers. Facebook introduced swap controls for the cgroupv2 mechanism and used it in the `fbtax2` project to improve system efficiency [10].

This trend highlights the importance of swap in modern systems. However, most of the current works focused on the

OS logic alone. Here we present a thorough analysis of the Linux swap performance focusing on the interplay between the swap logic and the SSD behavior.

3.1 Performance anomalies of swap on SSDs

3.1.1 GC is not aware of deallocated swap-slots

As shown in [Figure 1](#), the swap bandwidth decreases as the swapped-out data occupies a larger part of the device. In general, this behavior is expected because the GC overheads grow proportionally to the amount of actively updated data. However, the drop should not occur when a device is almost empty (occupied only 10% of its capacity).

The root cause is that the device-side GC is *not aware* that the OS discards some swapped-out pages and *invalidates* their respective swap-slots because the OS does not by default notify the SSD. Therefore, the *actual* occupancy of the swap device is much higher than the one visible to the OS, leading to higher GC overheads.

To cope with this issue, most SSDs implement a `TRIM` command that allows the OS to *hint* to the SSD to reclaim the storage of invalidated swap-slots. However, in practice, popular Linux distributions (e.g., Debian, Ubuntu) disable the use of `TRIM` command for swap [7, 9, 15, 21]. The reasons include `TRIM` dispatching overheads, the long latency of the `TRIM` command, and the complexity of supporting asynchronous `TRIMs` [35, 39, 50, 54, 54].

When `TRIMs` for swap are explicitly enabled, Linux issues the command *once* for a batch (*cluster*) of 512 invalidated swap-slots, to reduce the overheads. Notably, these swap-slots must be *contiguous* in the LBA address space [1].

To see the performance effect of `TRIMs`, we run the same random-write `vm-scalability` benchmark as in [Figure 1](#), but with `TRIMs` enabled (see § 6 for the setup). We measure the swap-out bandwidth and WAF over time as the device is being used to illustrate gradual performance degradation.

[Figure 2](#) shows that `TRIMs` (512-slot) have negligible effect. This is because the LBA contiguity requirement of `TRIM` clusters in Linux effectively prevents issuing `TRIMs` for the majority of the invalidated slots. These results corroborate the note in the `swapon` man page [20] that enabling `TRIMs` often does not improve swap performance.

Finer-grain `TRIMs` are not effective either. To demonstrate this, we develop a special mechanism that enables `TRIMs` for small contiguous clusters of eight swap-slots. This is not a practical approach, however, due to its high overheads (see § 6.1.1) [Figure 2](#) shows slight performance improvement, but still $2\times$ lower than the maximum bandwidth. Clusters smaller than 8 slots result in a prohibitively high rate of `TRIMs`, so the SSD cannot keep up with the swap-slot invalidation rate.

Observation 1: `TRIMs` are not effective at lowering GC overheads for swap.

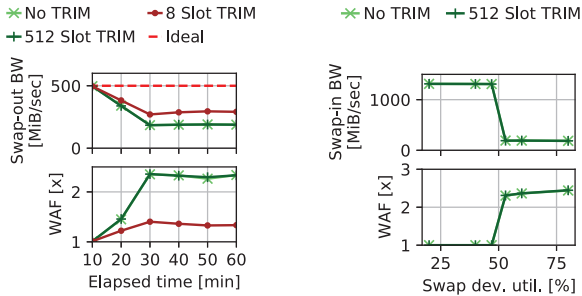


Figure 2: Swap-out bandwidth over time. Random memory writes using 40% of swap capacity.

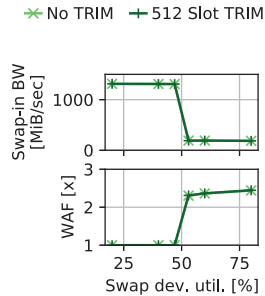


Figure 3: Swap-in bandwidth of random reads as a function of swap capacity utilization.

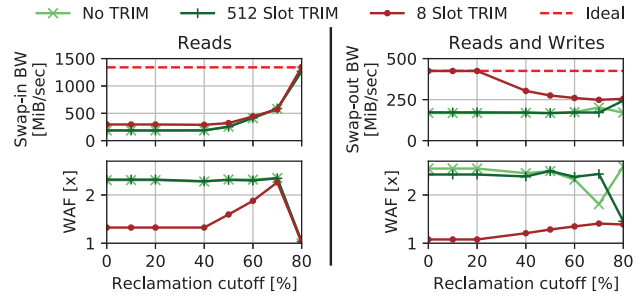


Figure 4: Swap-in and swap-out bandwidth for random reads and mixed reads and writes workloads respectively for different swap reclamation cutoffs.

3.1.2 Swap cache is not aware of GC

We execute the `vm-scalability` benchmark to perform uniform random reads on a large chunk of memory exceeding physical RAM and measure the swap performance for different values of the swap device utilization. Ideally, we expect the read performance to be independent of the utilization. Instead, Figure 3 shows a $6.9\times$ slowdown and $2.5\times$ WAF above 50% occupancy.

Our analysis shows that this problem occurs due to the way Linux implements its swap-cache. Recall that this cache is comprised of pages that are swapped into memory but the OS still maintains a copy in the swap. When the swap device’s utilization exceeds 50% – a hard-coded static parameter we call *swap reclamation cutoff*, Linux stops adding newly swapped-in pages to the swap-cache, invalidating their swap-slots immediately. As a result, the swap-out penalty for such pages incurs writing a page to the swap device, rather than discarding them from memory if they were in the swap-cache.

We suggest two possible reasons for this implementation. First, as the swap device gets full, the swap-slot allocation algorithm scans the swap-slot array linearly, which becomes slow [6]. Second, in the context of SSDs, deferring the swap-slot invalidation for in-memory pages effectively increases the device occupancy and eventually reduces performance due to the GC.

Unfortunately, the swap reclamation cutoff establishes a trade-off between the swap-out performance (preferring higher cutoff), and WAF (preferring lower cutoff). To illustrate, we measure the performance of two applications: one performing reads, and the other mixing both reads and writes. This setup aims to show that lower values of the reclamation cutoff are good for write-intensive workloads and bad for read-intensive ones. Higher values mirror this behavior.

We execute `vm-scalability` configured to use 80% of the swap device’s capacity. Half of the working set fits in RAM. Figure 4 shows the swap-in and swap-out bandwidth and WAF as a function of the swap reclamation cutoff. For random reads, the swap-in performance increases with the reclamation

cutoff, as fewer pages need to be written back upon eviction, with all the pages having copies both in the swap and in memory at the extreme. For the mixed workload, the effect of the cutoff is not visible with the default Linux configuration because the performance is low anyway. But with fine-grain TRIMs and higher baseline performance, smaller cutoff values are preferable.

Observation 2: The static reclamation cutoff strives to strike a balance between read and write performance, but instead aggressively prioritizes write workloads when the swap occupancy grows.

3.1.3 GC is not aware of page access pattern

We evaluate the performance of workloads with different memory access patterns using `pmbench`. We consider two write workloads: with uniform and with skewed access distributions (normal, $\sigma = \frac{1}{12}$ of the working set size, the default in `pmbench`). The swap-out bandwidth is 480MiB/sec (maximum for this SSD), and 195MiB/sec (WAF is 2.5) respectively, when using 5% of the swap capacity and 512-slot TRIMs enabled.

This difference stems from the different lifetimes of swapped-out pages. With the skewed distribution of memory writes, there are fewer opportunities for the swap subsystem to find large contiguous clusters of swap-slots to perform TRIMs, whereas uniformly distributed writes result in the swap-slots of similar lifetimes, increasing the chances of finding such clusters. 8-slot TRIMs are better, but the performance is still suboptimal: 324MiB/sec, with WAF of $1.5\times$.

Observation 3: Swap performance may vary significantly depending on the memory access pattern.

3.1.4 GC is not aware of OS’s performance isolation

Linux’s cgroup mechanism enforces resource isolation among different processes. In particular, it is possible to isolate swap bandwidth via `blk-throttle`. This is useful, e.g., in container-based virtualized environments to prevent performance interference between containers.

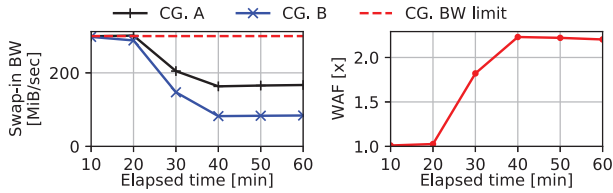


Figure 5: Swap-in bandwidth and WAF of 100%-random-read cgroup (A) and 50/50%-random-read/write cgroup (B) co-running together, each throttled to 300MiB/sec reads and 300MiB/sec writes.

We now evaluate the quality of the performance isolation in a scenario where we expect no interference. We run two processes, each in its own cgroup limited to 300MiB/sec reads and 300MiB/sec writes from/to the swap device. One process performs 100% reads and the other executes an equal mix of reads and writes, all uniformly distributed. To prevent any interference the processes are pinned to separate sets of cores, each with its own device queue. The aggregate bandwidth of the SSD does not reach its limit (1GiB/sec).

We expect both processes to achieve their maximum bandwidth allocation. In practice, during the first 20min of the execution (Figure 5) no GC is performed, thus the SSD sustains the cumulative request rate from both processes. When the GC is triggered, the swap-in bandwidth of *both* cgroups drops. Importantly, the first process performs only reads, and should not have been affected by the GC overheads caused only by the writes of the second process. This behavior stems from the GC’s inability to distinguish between the I/Os from different processes, and the OS’s inability to enforce bandwidth limits on the GC.

Observation 4: The GC impairs performance isolation dictated by the host OS.

3.2 Opportunities with swap on ZNS

ZNS SSDs provide better control over physical data placement, thereby enabling tighter coupling between the application logic and the device management, and have already been shown to offer new optimization opportunities for production Key-Value-Stores [25]. These results motivate a new GC-swap subsystem co-design that can leverage this coupling to mitigate the performance problems of traditional SSDs discussed above.

Is ZNS essential for performance? An important question is whether there is an inherent benefit to using ZNS SSDs over traditional ones, or one can redesign the swap subsystem alone to achieve the same outcome. In other words, can we achieve the performance of ZNS by emulating it on top of a Block SSD?

To answer, we run an experiment on a Block SSD while using a write access pattern that mimics the one enforced by ZNS zones. We run multiple threads, each performing

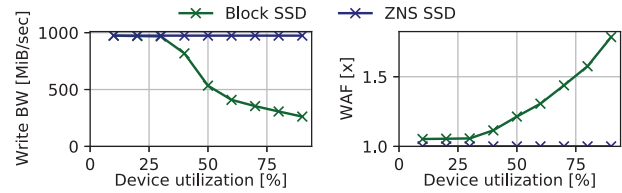


Figure 6: Write bandwidth and WA of sequential writes and TRIM operations to erase-block sized regions on Block SSD and ZNS SSD as a function of device utilization.

4KiB logically sequential writes with 1GiB-TRIMs (the size of an erase-block and a ZNS zone on our device). Each thread accesses its own part of a drive, and overwrites the available space, issuing a TRIM for the whole next 1GiB chunk. Multiple threads are used to emulate typical swap behavior.

We run the experiment for different values of device utilization. Figure 6 shows the results. We observe that the performance starts to decrease when a device is 30% full, drops to a half of the maximum bandwidth at 50%, and degrades down to a quarter at 80%. This is expected because the Block SSD cannot ensure that host-side TRIMs are aligned with physical erase-blocks as the writes from different threads get mixed in the device, even though the host strives to align them at the LBA level. In contrast, the same experiment on ZNS drives maintains full bandwidth no matter how full the device is.

We conclude that the *ZNS interface offers unique advantages that cannot be achieved with traditional Block SSDs.*

ZNS adoption. ZNS SSDs are expected to gain broader adoption in the near future. They hold the promise to reduce storage costs as they lower the internal DRAM size requirements, and might help reduce media overprovisioning via application-optimized software stack [25].

While the ZNS interface is not backward compatible with the in-place block interface, there is growing support for ZNS at the file system level. For example, F2FS and Btrfs in Linux can utilize ZNS drives.

These trends motivate us to tailor OS swap for ZNS SSDs.

4 Design

ZNSwap addresses three key design goals.

Resource-efficient Host-side GC. Reclaiming storage space in ZNS SSDs requires a host-side process akin to a GC that consolidates valid blocks from fragmented zones into new ones, subsequently erasing the freed zones. The primary challenge is in minimizing the memory and CPU overheads associated with the host-GC operation. This is because, unlike the device-side GC, the host-side GC directly competes for these host resources with regular applications. In essence, we need to *on-load* the GC to the CPU from the device with minimal costs, thereby enabling its tighter integration with the swap.

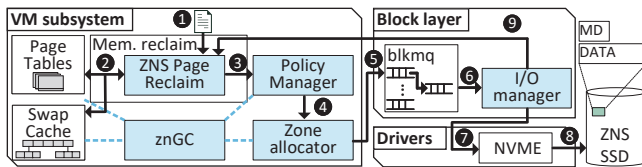


Figure 7: ZNSwap overview. Shaded shapes are internal ZN-Swap components.

These resource constraints preclude direct porting of existing host-side GC implementations. In particular, such implementations commonly maintain large translation tables (FTL) [32, 37], which consume about 1GiB for every TiB of data. The tables are frequently updated by writes and GC operations and accessed during reads. Given the typically poor locality of swap-induced I/O accesses [43, 55], these tables have to be resident in host memory. Maintaining the extra level of indirection between logical and physical block addresses appears to be inevitable to allow the host-side GC to move data without affecting the applications using it.

Our host-side GC, *znGC*, eliminates the need for the extra level of indirection entirely. It takes advantage of the fact that the swapped-out pages are still maintained in their owner’s page tables, and thus stores the relevant kernel reverse mapping metadata alongside the swapped-out page in the SSD. It also avoids I/O overheads to manage the reverse mappings by using the per-LBA metadata region available in NVMe devices as we describe in §§ 4.2 and 5.1.

ZNGC-OS integration. The key benefit of ZNGC over device-side GC is the ability to access the information exposed by the OS to optimize the swap I/O performance. For example, ZNGC may consult the OS-maintained swap-slot array to identify OS-invalidated swap-slots and avoid redundant copies without using TRIMs. We explain this and additional optimizations in § 4.3.

Swap data placement policies. Swap data placement may have a significant effect on the system performance, but the placement policy may depend on the specific execution environment. For example, a policy to achieve better resource isolation between a pair of processes might prefer storing all the pages of the same process in the same SSD zone. We strive to facilitate the implementation of such policies via a set of APIs that hide the complexity of zone management and ZNGC logic. We explain the API and the policies in § 4.3.

4.1 Overview

Figure 7 shows ZNSwap’s main design components. We explain each component and its role using the swap-out path as an example.

After a page to be swapped-out is selected by the OS, it is passed to the *ZNS page reclaim* ① which handles the page-table and swap-cache operations ②. In contrast to the original swap logic, it updates the destination location for the swapped-

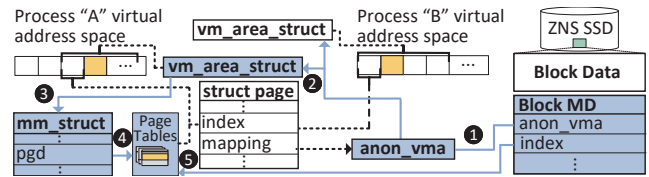


Figure 8: Linux reverse mapping overview. Shaded shapes are data structures accessed during ZNGC reverse mapping.

out page *after* it has been written, as dictated by the ZNS zone-append interface. Before writing a page, the page reclaim module consults the *policy manager* ③ which determines the destination *zone* and may guide ZNGC to free certain zones on the device. The policy manager incorporates custom policies that can be tailored to specific system requirements. The *zone allocator* ④ seamlessly handles Full zones and allocates a new zone when necessary.

The page is then submitted to the *block layer* ⑤, which subsequently passes the page to ZNSwap’s *I/O manager* ⑥. The I/O manager merges zone-append operations whenever possible, and generates an I/O request containing the page’s data and reverse mapping information used by ZNGC. Finally, the I/O manager hands off the I/O requests to the NVMe driver ⑦ which writes to the ZNS SSD ⑧, and updates the reclaim module with the page location on the SSD ⑨.

4.2 ZNGC

ZNSwap’s reclamation mechanism, ZNGC, is tightly integrated with the kernel virtual memory (VM) subsystem. ZNGC runs as a daemon in the kernel and is triggered either when the number of `Empty` zones is low, or via explicit requests by the ZNSwap policy.

Contrary to Block SSDs, a page moved by ZNGC is assigned a new host-visible address. Without an additional translation layer, ZNGC must update the page tables holding the original page swap-slot to reflect the new location. To this end, ZNGC stores the relevant reverse-mapping metadata alongside the data in the ZNS SSD’s per-LBA metadata region to assist later in updating the page tables. The storage interface (i.e., NVMe) allows to retrieve the metadata together with the respective data block in a single I/O operation. Thus, ZNGC retrieves the metadata to perform the reverse lookup of a given page and then updates the page table(s) that own it.

An important question remains: *which information* needs to be stored in the page metadata to guarantee that the reverse mapping remains correct during its lifetime?

To answer it, we leverage the same main data structures and procedures in the Linux kernel used to implement its reverse mapping scheme (Figure 8).

Background: Linux memory mapping structures. Recall that virtual memory pages in a process’ address space belong to virtual memory areas (vmas) that represent virtual memory allocations. vmas belong to a process’ virtual memory

address spaces (`mm_structs`), which hold the page table directory (`pgd`). The physical page descriptor (`struct page`) holds metadata enabling the reverse mapping. ZNGC stores the same metadata fields in the logical block’s metadata on the SSD.

To locate *all* page table entries associated with a physical page, the reverse mapping procedure accesses the `anon_vma` data structure, which is present between each physical page and the virtual memory area (`vma`) structures that map it¹. The `anon_vma` structure holds a list of `vmas` which may map the page and accounts for changes to the virtual mappings of the physical page. The physical page’s descriptor (`struct page`) does not directly account virtual mapping changes, rather, the descriptor holds a pointer to the `anon_vma` in its mapping field.

The `mm_structs` of each of the `vmas` that may map the page are accessed, and their page tables are walked to locate the page table entries. To calculate the virtual address used to walk the page tables, the `index` metadata value along with the `vma`’s start virtual memory address are used. The physical address corresponding to the physical page we have initialized the reverse mapping procedure is located in the last level page table entries and subsequently returned. Since swapped-out pages do not have a valid physical address in their page table entry, ZNGC returns the entries that hold the swapped-out location of the swap-slot we were performing the reverse mapping procedure.

Since the `anon_vma` structure is freed when there are no more `vmas` which may map the page and the `anon_vma` pointer in the `struct page` does not change, storing the pointer to the `anon_vma` as well as the mapping’s offset (`index`) within the logical block’s metadata on the SSD enables the same functionality as reverse mapping within the kernel.

4.3 ZNGC-swap integration

Physical zone information. Each zone is associated with a map of swap-slots. The map holds information on the use of each swap-slot, and whether it is valid, or swap-cached (similar to Linux’s `swap_map`). This information is used by ZNGC during the space reclamation. Note that a swap-slot can be discarded by the OS and ZNGC becomes immediately aware of the change, without using TRIMs as in Block SSDs. ZNGC may decide to reclaim some zones that are mostly free but hold some of the swap cache pages if it runs out of free storage space, making the swap reclamation cutoff parameter in Linux unnecessary.

Swap-zone abstraction. Active zones that can be used for swap-slot allocation are exposed via a *swap-zone* abstraction. A swap-zone is a virtual entity used to hide the complexity of managing physical SSD zones. Swap-zones are backed by Open zones. When an underlying physical zone transitions

¹anonymous pages and `vmas` only

Function	Purpose
<code>void rec_zn(int zn)</code>	Reclaim specified zone
<code>void pg_inf(pg_i*, u64 pfn)</code>	Get page statistics
<code>void vm_inf(vm_i*, u64 pfn)</code>	Get information on VMA
<code>void zn_inf(zn_i*, int zn)</code>	Get information on zone
<code>void swap_inf(swap_i*)</code>	Get ZNSwap statistics


```

typedef struct {
    u64 last_swapout_t;
    u16 access_bit_vec;
    int owner_pid;
    u64 cgroup_id;
} pg_i;

typedef struct {
    int zone_id;
    int capacity;
    int occupied_slots;
    int invalid_slots;
    int swap_cache_slots;
    int swap_zone_id;
} zn_i;

typedef struct {
    u64 vm_flags;
    u64 size;
    int readahead_win_sz;
    u64 cgroup_id;
} vm_i;

typedef struct {
    u64 num_{slots,zns};
    u64 free_{slots,zns};
    u8 zslot_array_sz;
    u32 {high,low}_wmark;
    bool gc_running;
} swap_i;

```

Table 1: ZNGC policy API.

to the Full state, it is seamlessly replaced by another Open physical zone. The total number of swap-zones is determined by the limit on the number of Open zones in the device.

ZNSwap policies. ZNSwap provides an API to facilitate the development of custom data placement and zone reclamation policies. A policy is invoked when the OS swaps-out a page, and its primary goal is to determine which swap-zone the page is written to. If there is a need to reclaim some of the zones, the policy may (asynchronously) invoke ZNGC to do so for a specific set of zones. The policies are implemented in a kernel module. Note that the swap-slot allocation policy operates at the granularity of swap-zones rather than swap-slots to conform to the ZNS interface.

API. A policy receives the page frame number (`pfn`) of the page being swapped-out and returns the `swap_zone_id` of the swap-zone where the swap-slot should be allocated. Table 1 lists the functions that can be invoked by the policy.

We define three sample policies:

per-core policy Attempts to assign a swap-zone per-CPU-core. If there are more cores than Open zones, the swap-zones are multiplexed. This mimics Linux’s swap-cluster per core policy and reduces contention on swap-zones.

hot/cold policy Utilizes a per-page access history bit-vector maintained by the OS and assigns hot and cold pages to different swap-zones.

cgroup policy Attempts to assign a swap-zone per-cgroup. If more cgroups are available, the swap-zones are multiplexed. If cgroup swap limits are set (max number of swap-slots), the policy will reclaim a zone used by the cgroup whose number of used zones exceeds the limit the most (as zones may contain invalidated swap-slots).

Example policy. We use cgroup policy to illustrate the use of the policy API. When invoked, the policy:

1. Selects the destination swap-zone for the cgroup (using

the `cgroup_id` from `pg_inf()`).

2. If the number of free physical zones is below a predefined low watermark (`swap_inf()`):
 - 2.1. Selects a victim cgroup whose number of utilized physical zones exceed its allocated swap-slot capacity the most.
 - 2.2. Iterates over the cgroup's physical zones (obtained via `swap_zone_id` from `zn_inf()` corresponding to the swap-zone allocation of the cgroup) and selects the zone with the least amount of valid slots.
 - 2.3. Triggers an asynchronous explicit reclaim on the victim zone (`rec_zn()`).
 - 2.4. Repeats the procedure until enough zones have been reclaimed (step 2.1).
3. Returns the destination swap-zone.

cgroup accounting. When a cgroup's swapped-out data is copied during the ZNGC operation, ZNGC's bandwidth is accounted as part of the cgroup's total bandwidth to the device. We do not yet implement the CPU accounting, but this is not critical as ZNGC's CPU overhead is low as we show in § 6.1.1.

4.4 Discussion

ZNSwap introduces the zoned namespace interface to core kernel mechanisms which used to support only traditional block devices. The ZNSwap's design is driven by the fundamental characteristics of ZNS SSDs, that are unattainable with traditional Block SSD, and which dictate the following design choices:

- **Zoned interface:** ZNSwap fully adheres to the zoned storage specification, therefore it inherits the specification's integral benefits. For example, ZNSwap utilizes zone-append operations to enable concurrent writes to sequential-write-only zones, accelerating the swap-out procedure to ZNS SSD by sequentially appending page data.
- **ZNS-related host responsibilities as opportunities:** ZNS requires implementing host-side GC, which present new opportunities for WA mitigation, better utilization of the SSD's capacity for swap-cache pages, and for improving performance isolation.
- **Tight integration of ZNSwap with kernel mechanisms:** utilizing fine granularity information the OS attains per swap-slot enables better synergy between the OS and ZNS SSD.

Hardware limitations. The number of possible destination zones for swapped-out pages in ZNSwap's data placement policies are limited by the number of `Open` zones the ZNS SSD supports, which is device specific. The limit affects the granularity of the policies' classifications. ZNSwap is designed to support ZNS SSDs with varying number of `Open` zones and zone sizes, and abstracts the intricacies of zone management via the swap-zone abstraction (§ 4.3).

ZNSwap also requires the use of the ZNS SSD's per-block metadata (64B). While per-block metadata is currently sup-

ported primarily in enterprise NVMe-SSDs, we believe that it will be a common feature among ZNS-SSDs.

5 Implementation

ZNSwap adds support for the zoned-interface model to key kernel mechanisms located in several subsystems. We implemented ZNSwap in Linux 5.12 with 4K LOC² (CLOC [8]).

5.1 ZNS page reclaim

Linux's reclamation algorithm is incompatible with the zone-append interface because it assumes that the write location of the swapped-out data is known before the write completion. Specifically, the algorithm uses the swap-slot as the key in the swap-cache for the page currently undergoing reclamation. If a page is accessed while it is being written to the swap device, a page-fault is raised, and the kernel locates the page in the swap-cache using the swap-slot entry to remap it.

ZNSwap redesigns the swap-out mechanism not to rely on the pre-acquired swap-slot entry. The main idea is to utilize the dirty bit of the page's page-table entry to indicate whether the page has been dirtied during the data transfer to the swap device. Write access to such a page does not raise a page-fault since the page is *still mapped* in the page-table. Rather, we check the dirty bit in the page-table when unmapping it. We provide more details in Appendix A.

5.2 ZNGC

We now describe the zone reclamation process in detail. ZNGC first selects a candidate zone from a preselected set of zones supplied by the policy. Given a zone, ZNGC scans through batches of pages until a whole zone is reclaimed. Figure 9 depicts the main stages:

Gather. ZNGC checks the swap-slots in the candidate zone. Swap-slots of the pages currently cached by the swap-cache are removed from the swap-cache and their swap-slots are invalidated. Occupied valid swap-slots are gathered into a pre-allocated array of block IOs to perform device reads. This stage completes when the block IO array is full, or until it reaches the end of the zone.

Read. The occupied blocks IOs containing the read operations are dispatched as a batch of requests to the device. The destination of each read operation corresponds to a page from a pre-allocated page pool. The metadata for each swap-slot is read from the device into a buffer.

Write. Once all read operations are complete, each occupied page from the page pool is examined and assigned a destination zone based on the ZNGC-swap policy. The block IO array is subsequently reused to hold all the pending write requests, which are dispatched as a batch.

²<https://github.com/acsl-technion/znszap>

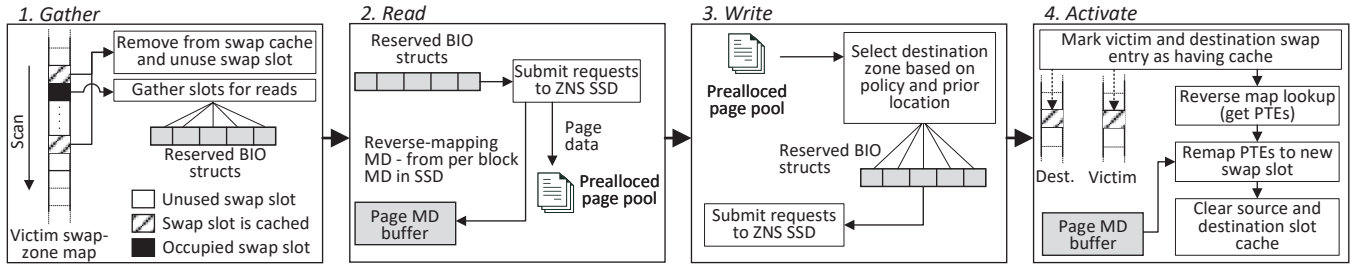


Figure 9: ZNGC: main stages in garbage-collecting a victim zone.

Activate. After the write operations complete, the corresponding swap-slots in the victim’s zone are re-examined. If a swap-slot is still valid and occupied, it is marked as if it resides in swap-cache in both victim and destination zones, to indicate to other kernel procedures that these swap-slots are currently in use. The page-table entries corresponding to the victim swap-slots are subsequently remapped to hold the destination swap-slots with the help of the reverse mapping information obtained from the metadata (`mapping` and `index`). Finally, the victim swap-slots are cleared. After ZNGC traversal over a zone is complete, the zone is reset.

Concurrent accesses to swapped-out pages undergoing migration trigger a regular swap-in operation. ZNGC will skip the corresponding swap-slot’s migration as the page resides in memory, and will continue with the reclamation of the zone.

ZNGC does not perform dynamic memory allocations and is designed to occupy a minimal amount of physical memory (up to 5MiB).

5.3 I/O manager

ZNSwap’s I/O manager adds support for zone-append merges and seamlessly stores the per-page reverse-mapping information into the metadata region of each written LBA.

Zone append merges. ZNGC and the page-out procedures take advantage of the `blk_plug` mechanism to batch together zone-append operations destined to the same zone. We add support for zone-append merges in the block layer by identifying block IOs destined towards the same zone that are waiting to be drained and merge the page-list of each block IO, creating a single block IO request. Once the request has been completed, we iterate over the pages in the request and generate an independent completion notification to each of the merged block IOs with their respective write location, calculated from their offset within the merged block IO and its final location.

Reverse mapping metadata. The I/O manager allocates a DMA-mapped physical page pool for metadata associated operations. The pages serve as a host buffer for the per-page metadata, and act either as a source or target location for append and read I/Os, respectively. The DMA address of the pages is supplied as part of the per-LBA metadata for each command. When serving a swap-out append operation, each

LBA stores 16 bytes of metadata for the reverse mapping information of the page (`mapping` and `index`).

6 Evaluation

Our evaluation demonstrates the benefits of ZNSwap using ZNS SSDs over the Linux swap using Block SSDs. In particular, we focus on the benefits of integrating the ZNGC with the host OS and the usefulness of ZNGC policies. We note that all our benchmarks perform direct *memory* accesses only, and impose SSD accesses due to the swap activity. Thus, the actual SSD access pattern might differ from the memory access pattern in the benchmark.

Can ZNS drives be used via compatibility layers? Linux swap does not work on top of ZNS drives, either as a swap-file or a swap partition. Existing Linux device-mappers such as `dm-zoned` [49] and `dm-zap` [33] aim to expose zoned devices as regular block devices without any write-pattern constraints but require large mapping structures for indirection. However, they do not currently support ZNS SSDs. Therefore, the only plausible baseline is Linux swap with block SSDs.

Hardware. We use a server with 2× Intel Xeon Silver 4216 CPU and 512 GiB of memory (2× 256 GiB DDR4 2933 Hz), with Ubuntu 20.04, Linux 5.12.0. HyperThreading is disabled, the frequency governor is "performance", and "turbo" is disabled to achieve stable results. We use a 1 TB production-grade Western Digital ZN540 ZNS SSD and an equivalent 1TB conventional Block SSD (with 7% OP) that uses the same hardware platform and media. Both SSD’s maximum sequential read and write bandwidth is 3.2 GiB/sec and 1 GiB/sec respectively. Random 4 KiB reads and writes reach 1.4 GiB/sec and 1 GiB/sec respectively. For the ZNS SSD, the writeable capacity of each zone is 1077 MiB, and is formatted with the ability to store 64 B of metadata per LBA.

Setup. We configure the swap size to be the size of the system memory (512 GiB) according to the common practice [19]. The remaining capacity on the drives is filled with data. The resulting effective OP of the swap partition in the Block SSD is 12%, therefore we configure ZNSwap to use the same OP.

Before each experiment, the SSD is formatted, followed by a ramp-up until the workload has reached its steady state [17]. Bandwidth and WAF measurements are sampled at 10 min

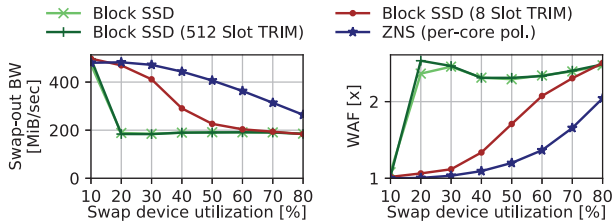


Figure 10: Swap-out bandwidth of `vm-scalability` with random memory writes. As expected, higher device utilization results in higher GC load.

intervals. The Block SSD’s WAF is measured through the device’s internal host- and media-writes counters, and the ZNS SSD’s WAF is measured by recording the number of writes performed by ZNGC.

Performance metrics and optimal performance. We primarily focus on the swap-out bandwidth as the main performance metric. The rationale is that under write-intensive memory access pattern, swap-in operations trigger the eviction of an equal amount of dirty pages to the drive. Hence, the resulting SSD access pattern is an equal mixture of 4KiB random reads and mostly random writes for Block SSD, and random reads and sequential writes for ZNS SSD.

The maximum write bandwidth for such a 50%/50% access pattern on both Block SSD and ZNS SSD drives is measured to be 488 MiB/sec. Therefore, we claim that the ZNSwap’s performance benefits over the Linux swap baseline presented in this section *stem primarily from ZNSwap design rather than from the performance differences among the drives or the difference in the access patterns.*

6.1 Synthetic benchmarks

We use the standard swap performance benchmarks, `vm-scalability` [22] and `pmbench` [58] which allow evaluating the performance of the swap subsystem and the swap device under different memory access patterns.

We rerun several experiments from the Motivation section on ZNSwap, to show how it recovers the system performance for the cases where the standard Linux swap on Block SSDs suffers from the performance anomalies.

6.1.1 Benefits of ZNGC-swap subsystem integration

Swapping without TRIMs. We execute `vm-scalability` in a 2 GiB memory-limited cgroup. In each experiment, we pre-allocate different amounts of memory to evaluate different levels of the swap device’s capacity utilization. We then perform random writes to that memory (`case-anon-w-rand-mt`), resulting in random read/writes from/to the swap device. To maximize throughput, we execute 64 threads ($2\times$ the number of available cores). This is the same experiment as in § 3.1.1.

Figure 10 shows the results. ZNSwap immediately observes the OS-managed swap-slot allocation without using TRIMs, and as such only moves the valid pages when running ZNGC. While ZNGC adds overheads to the host, ZNSwap outperforms the Linux swap in all cases but at 10% utilization due to the device WA being lower.

CPU overheads of ZNGC vs. fine-grain TRIMs with Block SSD. We measure the maximum CPU overhead of ZNGC under 80% swap device utilization in Figure 10. We observe that ZNGC occupies 15% of a single CPU core. At 10% swap device utilization, the overhead drops to a negligible 0.3%. In contrast, the CPU overhead for dispatching 8-slot TRIMs is 32% of a single CPU core with lower swap performance compared to ZNSwap.

Swapping without swap reclamation cutoff. We run the same experiment with read-only and mixed read-write benchmarks as in § 3.1.2 where we established the performance degradation due to the static swap reclamation cutoff in the standard swap. When invoked with ZNSwap, the performance matches the “ideal” line in Figure 4.

6.1.2 Skewed workloads

We run `pmbench` configured to allocate 320 GiB of memory and perform skewed memory writes with the default normal distribution parameters ($\sigma = \frac{1}{12}$ of the allocated memory). The distribution directs 80% of the memory accesses to 20% of the allocated memory considered “hot”. The other 80% of the allocated “cold” memory occupies 50% of the swap capacity. The “hot” pages’ lifetime in swap tends to be shorter than for other pages. In each experiment, we modify the amount of RAM available to `pmbench` thus varying the proportion of the working set swapped-out from 50% to 90%. This allows to vary the swap device utilization without changing the working set size and page access pattern across the experiments.

We compare the baseline with ZNSwap with the per-core policy that ignores the page access frequency, and ZNSwap with the hot/cold policy that strives to group pages with similar access frequencies into the same zone (see § 4.3).

We make two observations. First, ZNSwap achieves the same performance regardless of the access pattern up to $2\times$ higher bandwidth compared to the baseline for both ZNSwap policies. Second, the hot/cold policy exhibits 15-20% lower WA compared to the naive policy, even though this benefit is not reflected in the swap-out bandwidth in this workload. Reducing WA is important on its own to achieve a higher device lifetime [31].

6.1.3 Swap performance isolation in cgroups

We execute two instances of the `vm-scalability` benchmark, each in a different cgroup. Both cgroup A (CG. A) and cgroup B (CG. B) run with 16 threads. CG. A utilizes 30% of the swap capacity, and performs random *writes*, whereas

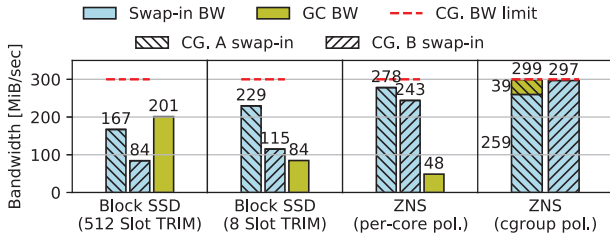


Figure 11: Bandwidth distribution among different cgroups, one reading and another writing data.

CG. B utilizes 10% of the swap capacity and performs only random *reads*. In addition, A’s and B’s swap bandwidth is limited via `blk-throttle` to 300 MiB/sec. This is the same experiment as in § 3.1.4.

Figure 11 shows the swap-in bandwidth for each cgroup under different configurations. If the swap performance isolation was perfect, each cgroup would behave as if it runs with its own SSD, reaching its target bandwidth (red line). With Block SSD, however, the target bandwidth cannot be attained. Figure 5 shows that without the device-GC overhead, the upper bound is reached, implying that in this experiment this overhead indeed causes performance degradation. Similar to Block SSD, ZNS SSD with ZNSwap’s per-core placement policy fails to provide swap isolation.

In contrast, with ZNSwap’s cgroup policy, the bandwidth of the writer process (CG. A) fully absorbs the ZNGC bandwidth overheads because only the data attributed to that cgroup is moved by ZNGC. Note that the sum of the bandwidth used by the swap and ZNGC operations in that cgroup does not exceed the predefined limit. CG. B attains full bandwidth, and it is not affected by the ZNGC bandwidth overheads.

6.1.4 Raw swap performance

We stress-test the raw swap-out performance of ZNSwap and its multi-core scaling. We execute `vm-scalability` to sequentially write (`case-anon-w-seq-mt`) 500 GiB of data in a contiguous memory region, while limiting the memory size to 2 GiB via cgroup. By choosing the sequential access pattern, not reusing the same pages, and limiting the number of writes to not surpass the device’s capacity, we force the system to avoid reusing swap-slots thus preventing device-side GC and swap-in operations. This is done to achieve the highest performance, stressing the swap software mechanisms.

ZNSwap exhibits the same performance as the traditional Linux swap, achieving 740 MiB/sec swap-out bandwidth for a single core, and the maximum device bandwidth of 1 GiB/sec with 4 cores (no graph shown).

6.2 End-to-end application Benchmarks

We evaluate two popular key-value store servers, demonstrating the benefits of ZNSwap to run large-memory produc-

tion applications. The throughput and latency we obtain are consistent with those reported for other flash-assisted KVS works [36, 46, 57].

We execute the KV servers on one NUMA node, and the client on the other; hence we set the affinity of both NUMA nodes’ `kswapd` threads as well as the `kznsd` thread that executes ZNGC to run on the first NUMA node to co-locate them with the application. Thus, both ZNSwap and traditional Linux swap are allocated *the same* amount of compute resources which they share with the application threads.

6.2.1 memcached-ETC

We run a `memcached` key-value store [29] using the `mutilate` client [45] and Facebook’s ETC benchmark [23]. We evaluate a random-skewed access pattern with 90% of requests accounting for 10% of the keys. Despite this skewness, the distribution of popular keys in the *memory* is mostly uniformly random because they are scattered across different memory pages. This also dictates random access to the SSD.

We configure `memcached` to use 32 threads on one NUMA node, and invoke 32 `mutilate` client threads on the other NUMA node. We load the data to the server until we reach the target swap device capacity utilization. We do not limit the amount of memory available to the server, thus utilizing all memory (from both NUMA nodes) for the workload. For example, 10% swap utilization (51 GiB) implies the total working set of 563GB. We report the 99p latency of the KV store, maximum throughput, as well as the WAF of the SSD.

Figure 12 shows that ZNSwap consistently outperforms Block SSD-based swap in all performance metrics under the evaluated swap device utilization: under 10% swap device utilization ZNSwap exhibits 10× lower 99p latency and 5× higher maximum QPS while not experiencing any WA, as opposed to Block SSD which suffers from a 2.5× WAF. With the added 8 blk. TRIM support for Block SSD, ZNSwap achieves 5× lower 99p, 1.6× higher QPS and 1.1× lower WAF.

6.2.2 redis-YCSB

We use an in-memory `redis` data store [16] with the `YCSB` client [27] configured with 50% reads and 50% updates (update-heavy configuration) in a 20-80 hotspot distribution (80% of accesses target 20% of the working set) which is one of the standard options. This memory access pattern induces the same distribution of accesses as we evaluated in § 6.1.2, thereby allowing us to show the application performance impact of the hot/cold placement policy.

`redis` is executed in cluster-mode consisting of 32 server-threads, running on one NUMA node in a RAM-limited cgroup. It loads a 320 GiB dataset to the cluster. 64 client threads are spawned on the other NUMA node. Similar to the microbenchmark in § 6.1.2, we vary the amount of available RAM while keeping the working set size constant.

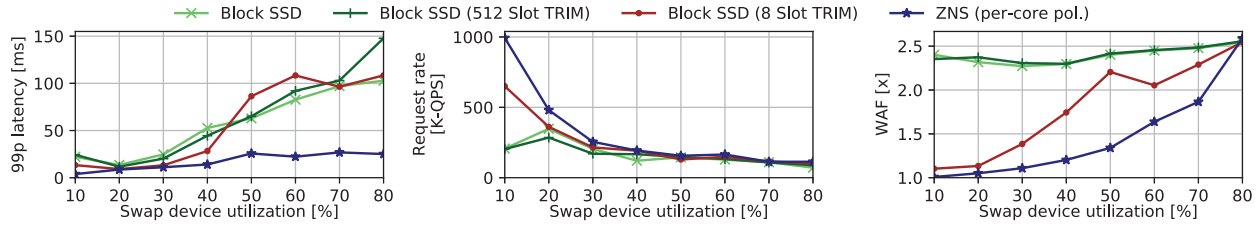


Figure 12: memcached Facebook ETC 99 percentile latency at the highest throughput

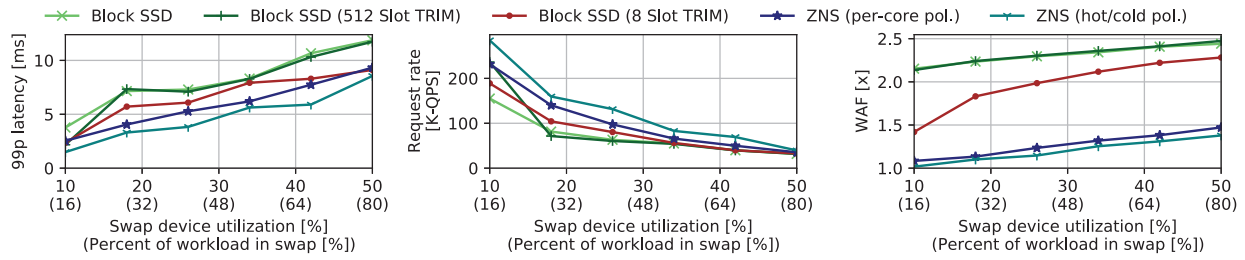


Figure 13: redis 20-80 hotspot distribution 50/50 read/write, 99p latency at maximum throughput

Figure 13 shows the 99p latency, throughput and WA for ZNSwap’s per-core and hot-cold policies, as well as Block SSD. Both ZNSwap’s policies outperform Block SSD in all performance metrics. We observe a $1.27\times$ speedup in throughput and $1.4\times$ drop in latency with $1.1\times$ lower WA for ZNSwap’s hot/cold policy compared to the per-core policy.

7 Related work

SSD-friendly swap support. SSDs’ unique characteristics warranted a body of works [48, 54] that aim to optimize swap on Block SSDs. These works modify Linux’s page reclaim policy (similar to CFLRU [52]) to prioritize reclamation of clean pages and reduce device-side GC overheads without modifying the GC itself. In contrast, ZNSwap offers a novel co-design of the host-side GC and the swap mechanisms and achieves its benefits via tighter coupling between them.

Swap on raw flash. Several early works proposed swap to raw flash [38, 41, 47] thereby avoiding GC overheads due to copying blocks of discarded swap-slots. These papers predated the introduction of native TRIM support in SSDs, which was supposed to achieve the same effect. ZNSwap shows that even fine-grain TRIMs are not sufficient, and demonstrates other benefits of the tight coupling with the OS enabled by the host-side GC.

Open-channel SSDs [26] expose a low-level storage management interface, similarly to ZNS. ZNSwap’s main contribution is its study of the benefits of host-side SSD management and swap co-design, not considered in prior works. Further, unlike ZNS, the adoption of OC-SSDs so far has been limited due to poor portability and the complexity of the host-side media control they require, such as media wear-levelling.

Stream-SSDs [40] expose a traditional block interface, and can reduce WA by utilizing hints so the device may attempt

to co-locate data with similar lifetimes onto the same erase-blocks. However, Stream-SSDs’ block-interface hinders support for cross-layer optimizations introduced by ZNSwap on ZNS-SSDs, which are key to ZNSwap’s performance gains.

Implementing swap data placement support for Stream-SSDs, akin to ZNSwap’s swap policies, will offer certain benefits in the scenarios where data lifetime can be predicted and data consolidated into a set number of streams, such as hot/cold access patterns (as noted in § 6.1.2). However, under random access patterns, Stream-SSDs would perform similarly to traditional Block SSDs. The performance gains pertaining to ZNSwap’s cross-layer optimizations that aren’t related to data-placement policies (i.e., the elimination of TRIMs) exhibit higher performance gains than data-placement policies, as shown in Figure 10.

8 Conclusion

ZNSwap leverages the recent ZNS SSD interface to enable tight integration of the storage management mechanisms with the swap subsystem. ZNSwap introduces a host-side ZNGC that is co-designed with the swap logic to reduce garbage-collection overheads and improve system performance, while also leveraging the tight coupling with the OS and NVMe metadata interface to avoid the costly flash translation layer in the host. ZNSwap demonstrates significant performance advantages of using ZNS for swap in realist scenarios, paving the way to broader adoption of this new technology.

Acknowledgements

We gratefully acknowledge support from Israel Science Foundation (grants 980/21 and 1027/18) and financial support from Western Digital.

References

- [1] Swapfile: swap allocation use discard. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7992fde72ce06c73280a1939b7a1e903bc95ef85>, 2009.
- [2] Making swapping scalable. <https://lwn.net/Articles/704478/>, 2016.
- [3] Reconsidering swapping. <https://lwn.net/Articles/690079/>, 2016.
- [4] NVM Express 2.0 Zoned Namespace Command Set Specification. <https://nvmexpress.org/specifications>, 2018.
- [5] SAMSUNG. Ultra-low latency with Samsung Z-NAND SSD. http://www.samsung.com/us/labs/pdfs/collateral/Samsung_ZNAND_Technology_Brief_v5.pdf, 2019.
- [6] Swap: try to scan more free slots even when fragmented. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ed43af10975eef7e21abbb81297d9735448ba4fa>, 2020.
- [7] Archlinux SSD Optimizations. https://wiki.archlinux.org/title/Solid_state_drive#Continuous_TRIM, 2021.
- [8] cloc: Count lines of code. <https://github.com/AlDanial/cloc>, 2021.
- [9] Debian SSD Optimizations. https://wiki.debian.org/SSDOptimization#Mounting_SSD_filesystems, 2021.
- [10] Facebook cgroupv2 memory controller. <https://facebookmicrosites.github.io/cgroup2/docs/memory-controller.html>, 2021.
- [11] Kioxia's PCIe 5.0 SSD Just Hit 14,000 MBps. <https://www.tomshardware.com/news/kioxia-pcie-5-ssd-just-hit-140000-mbps>, 2021.
- [12] Memcg backend asynchronous reclaim. <https://partners-intl.aliyun.com/help/doc-detail/169535.htm>, 2021.
- [13] Multi-generational LRU: the next generation. <https://lwn.net/Articles/856931/>, 2021.
- [14] OpenStack: Overcommitting CPU and RAM. <https://docs.openstack.org/arch-design/design-compute>, 2021.
- [15] Red Hat: Discarding Unused Blocks. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/managing_file_systems/discarding-unused-blocks_managing-file-systems, 2021.
- [16] Redis. <https://redis.io>, 2021.
- [17] Solid State Storage Performance Test Specification. https://www.snia.org/sites/default/files/technical_work/PTS/SSS_PTS_2.0.2.pdf, 2021.
- [18] Swap file on Amazon EC2. <https://aws.amazon.com/premiumsupport/knowledge-center/ec2-memory-swap-file/>, 2021.
- [19] Swap space on Amazon EC2. <https://aws.amazon.com/premiumsupport/knowledge-center/ec2-memory-partition-hard-drive/>, 2021.
- [20] swapon(8) Linux man pages. <https://man7.org/linux/man-pages/man8/swapon.8.html>, 2021.
- [21] Ubuntu: TRIM the swap partition. <https://wiki.ubuntuusers.de/SSD/TRIM/#TRIM-der-Swap-Partition>, 2021.
- [22] vm-scalability. <https://git.kernel.org/pub/scm/linux/kernel/git/wfg/vm-scalability.git>, 2021.
- [23] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.
- [24] Matias Bjørling. Zone Append: A New Way of Writing to Zoned Storage. In *Vault Linux Storage and Filesystems Conference*, Santa Clara, CA, February 2020. USENIX Association.
- [25] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, DL Moal, G Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*, 2021.
- [26] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies FAST 17*, pages 359–374, 2017.
- [27] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

- [28] Peter Desnoyers. Analytic models of SSD write performance. *ACM Transactions on Storage (TOS)*, 10(2):1–25, 2014.
- [29] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.
- [30] Javier González, Matias Bjørling, Seongno Lee, Charlie Dong, and Yiren Ronnie Huang. Application-driven flash translation layers on open-channel SSDs. In *Proceedings of the 7th non Volatile Memory Workshop (NVMW)*, pages 1–2, 2016.
- [31] Laura M Grupp, John D Davis, and Steven Swanson. The bleak future of NAND flash memory. In *FAST*, volume 7, pages 10–2, 2012.
- [32] Aayush Gupta, Youngjae Kim, and Bhuvan Uргаonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. *ACM SIGPLAN Notices*, 44(3):229–240, 2009.
- [33] Hans Holmberg. dm-zap: Host-based FTL for ZNS SSDs. <https://github.com/westerndigitalcorporation/dm-zap>, 2021.
- [34] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–9, 2009.
- [35] Choulseung Hyun, Jongmoo Choi, Donghee Lee, and Sam H Noh. To TRIM or not to TRIM: Judicious trimming for solid state drives. In *Poster presentation in the 23rd ACM Symposium on Operating Systems Principles*, 2011.
- [36] Junsu Im, Jinwook Bae, Chanwoo Chung, Sungjin Lee, et al. Pink: High-speed in-storage key-value store with bounded tails. In *2020 USENIX Annual Technical Conference (USENIX ATC’ 20)*, pages 173–187, 2020.
- [37] Song Jiang, Lei Zhang, XinHao Yuan, Hao Hu, and Yu Chen. S-FTL: An efficient address translation for flash memory by exploiting spatial locality. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2011.
- [38] Dawoon Jung, Jin-soo Kim, Seon-yeong Park, Jeong-uk Kang, and Joonwon Lee. Fass: A flash-aware swap system. In *Proc. of International Workshop on Software Support for Portable Storage (IWSSPS)*. Citeseer, 2005.
- [39] Dong Hyun Kang and Young Ik Eom. TO FLUSH or NOT: Zero padding in the file system with SSD devices. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–9, 2017.
- [40] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *6th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.
- [41] Sohyang Ko, Seonsoo Jun, Yeonseung Ryu, Ohhoon Kwon, and Kern Koh. A new linux swap system for flash memory storage devices. In *2008 International Conference on Computational Sciences and Its Applications*, pages 151–156. IEEE, 2008.
- [42] Gyun Lee, Wenjing Jin, Wonsuk Song, Jeonghun Gong, Jonghyun Bae, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. A case for hardware-based demand paging. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 1103–1116. IEEE, 2020.
- [43] Jaehun Lee, Sungmin Park, Minsoo Ryu, and Sooyong Kang. Performance evaluation of the SSD-based swap system for big data processing. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 673–680. IEEE, 2014.
- [44] Jongsung Lee and Jin-Soo Kim. An empirical study of hot/cold data separation policies in solid state drives (SSDs). In *Proceedings of the 6th International Systems and Storage Conference*, pages 1–6, 2013.
- [45] Jacob Leverich. Mutilate: high-performance memcached load generator, 2014.
- [46] Cheng Li, Hao Chen, Chaoyi Ruan, Xiaosong Ma, and Yinlong Xu. Leveraging NVMe SSDs for building a fast, cost-effective, LSM-tree-based KV Store. *ACM Transactions on Storage (TOS)*, 17(4):1–29, 2021.
- [47] Mingwei Lin and Shuyu Chen. Flash-aware linux swap system for portable consumer electronics. *IEEE Transactions on Consumer Electronics*, 58(2):419–427, 2012.
- [48] Mingwei Lin, Shuyu Chen, and Guiping Wang. Greedy page replacement algorithm for flash-aware swap system. *IEEE Transactions on Consumer Electronics*, 58(2):435–440, 2012.
- [49] Damien Le Moal. dm-zoned: Zoned Block Device device mapper. <https://lwn.net/Articles/714387/>, 2017.
- [50] Trong-Dat Nguyen and Sang-Won Lee. I/O characteristics of MongoDB and trim-based optimization in flash SSDs. In *Proceedings of the Sixth International Conference on Emerging Databases: Technologies, Applications, and Theory*, pages 139–144, 2016.

- [51] Ohshima, S. Scaling flash technology to meet application demands. Keynote 3 at Flash Memory Summit 2018, 2018.
- [52] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jinsoo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 234–241, 2006.
- [53] SeongJae Park, Yunjae Lee, Moonsub Kim, and Heon Y Yeom. Automating context-based access pattern hint injection for system performance and swap storage durability. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.
- [54] Mohit Saxena and Michael M Swift. FlashVM: Virtual Memory Management on Flash. In *USENIX Annual Technical Conference*, 2010.
- [55] Taejoon Song, Gunho Lee, and Youngjin Kim. Enhanced flash swap: Using NAND flash as a swap device with lifetime control. In *2019 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–5. IEEE, 2019.
- [56] Benny Van Houdt. Performance of garbage collection algorithms for flash-based solid state drives with hot/cold data. *Performance Evaluation*, 70(10):692–703, 2013.
- [57] Shuotao Xu. *Bluecache: A scalable distributed flash-based key-value store*. PhD thesis, Massachusetts Institute of Technology, 2016.
- [58] Jisoo Yang and Julian Seymour. Pmbench: A micro-benchmark for profiling paging performance on a system with low-latency SSDs. In *Information Technology-New Generations*, pages 627–633. Springer, 2018.
- [59] Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. FlashKV: Accelerating KV performance with open-channel SSDs. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–19, 2017.

A Pageout process

Figure 14 illustrates the operations performed during the pageout process in detail.

Traditional page-out. A candidate anonymous memory page from the inactive-list [1] is selected to be evicted (not recently accessed [2]) and is not in the swap cache [3], it is assigned a swap-slot entry [4]. The swap-slot entry is used both as the destination of the page in the swap device, as well as its identifier within the swap-cache. After the page has been inserted into the swap-cache and subsequently unmapped from the page tables [6], the swap-slot entry value is inserted instead. If the page is dirty [7], it is unmarked as such, the write operation to the swap device initiates [8], and the page is reinserted into the head of the inactive list [9].

After the page has been successfully written to the swap device, it is moved to the tail of the inactive list [10], where it is then removed for the second time [11] and passes through the same conditions as in the first iteration. Finally, the page is freed along with its swap-cache entry [16].

If the page is accessed during the write to the swap device, it is located in the swap-cache using the swap-slot entry, and will subsequently fail one of the conditions in [12-15].

ZNSwap page-out. Apart from sampling the accessed bit in [2], the dirty bit is sampled, cleared, and stored in the `PG_dirty` flag of the `struct page`. The page is then assigned a zone per the defined policy [4] and the append operation to the swap device initiated [5]; the page is then reinserted to the inactive-list [6]. Once the append operation has been completed and the location of the written data retrieved, the page is inserted into the swap-cache. The `PG_dirty` flag is cleared and the page is moved to the tail of the inactive-list [7]. The page then traverses through [8-11] and is unmapped from the page tables [13]. If the page has been dirtied since the append operation has initiated [14-15], the page-out operation is aborted. The page is finally freed at [16].

Unlike the traditional page-out algorithm, an access to the page while it is undergoing write-back to the swap device will not raise a page-fault and subsequently be remapped since it is *still mapped* in the page-tables. Rather, the dirty bit in the page tables is evaluated during the unmapping process [14], which indicates whether it is safe to free the page or not.

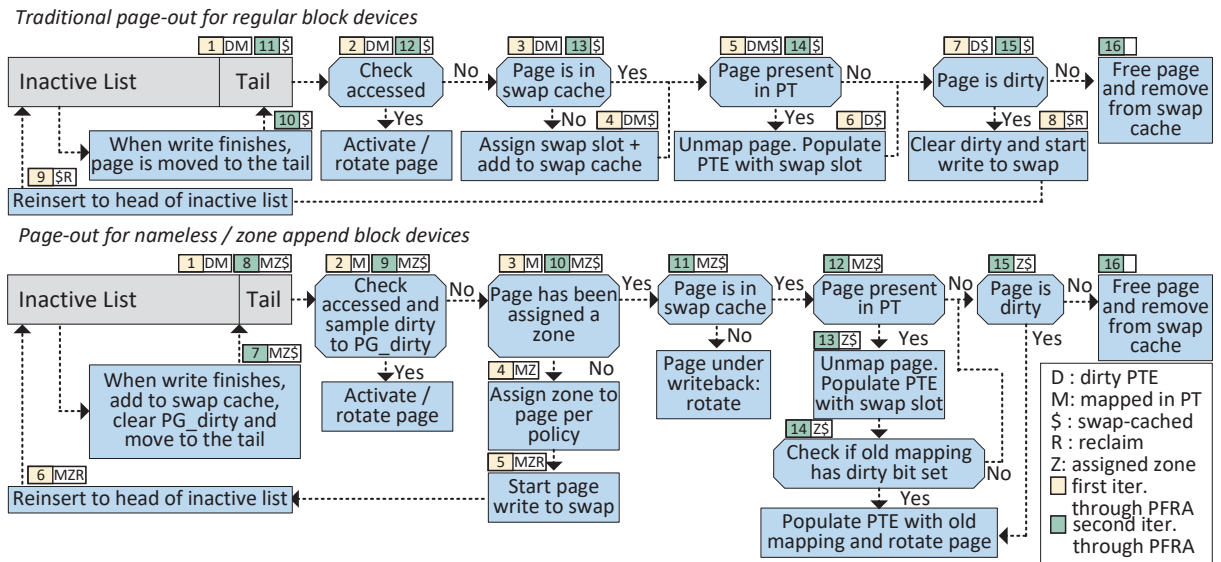


Figure 14: Page-out procedure for inactive pages

Building a High-performance Fine-grained Deduplication Framework for Backup Storage with High Deduplication Ratio

Xiangyu Zou[†], Wen Xia[†], Philip Shilane^{*}, Haijun Zhang[†], and Xuan Wang[†]

[†] Harbin Institute of Technology, Shenzhen ^{*} Dell Technologies

Corresponding author: xiawen@hit.edu.cn

Abstract

Fine-grained deduplication, which first removes identical chunks and then eliminates redundancies between similar but non-identical chunks (i.e., delta compression), could exploit workloads' compressibility to achieve a very high deduplication ratio but suffers from poor backup/restore performance. This makes it not as popular as chunk-level deduplication thus far. This is because allowing workloads to share more references among similar chunks further reduces spatial/temporal locality, causes more I/O overhead, and leads to worse backup/restore performance.

In this paper, we address issues for different forms of poor locality with several techniques, and propose MeGA, which achieves backup and restore speed close to chunk-level deduplication while preserving fine-grained deduplication's significant deduplication ratio advantage. Specifically, MeGA applies ① a backup-workflow-oriented delta selector to address poor locality when reading base chunks, and ② a delta-friendly data layout and "Always-Forward-Reference" traversing in the restore workflow to deal with the poor spatial/temporal locality of deduplicated data.

Evaluations on four datasets show that MeGA achieves a better performance than other fine-grained deduplication approaches. In particular, compared with the traditional greedy approach, MeGA achieves a 4.47–34.45 \times higher backup performance and a 30–105 \times higher restore performance while maintaining a very high deduplication ratio.

1 Introduction

Chunk-level deduplication [2, 7, 18, 20, 27, 28, 40, 45, 54] has been widely used in backup storage systems to reduce storage costs, but it is limited by its coarse-grained processing granularity (i.e., file/chunk level) and can not completely exploit data workloads' compressibility. To achieve a higher deduplication ratio, fine-grained deduplication [22, 38, 47] is proposed.

Fine-grained deduplication, sometimes previously called "delta compression," not only focuses on duplicate chunks but also removes sub-chunk-level redundancies existing in

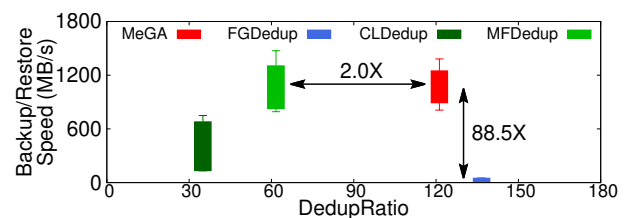


Figure 1: Performance of MeGA (our approach), FG Dedup (a typical fine-grained deduplication approach similar to SDC [53]), CL Dedup (a typical chunk-level deduplication approach [23]), and MF Dedup (a special chunk-level deduplication approach [58]) on a website snapshot dataset.

similar but non-identical chunks, and it has been studied in several use cases [15, 37, 38, 51]. Typically, fine-grained deduplication first deduplicates identical chunks, then finds similar base chunks (among non-duplicates), and finally runs delta encoding between the new and base chunks to only store their differences (a.k.a., delta chunks) for space-saving. As a result, fine-grained deduplication could achieve a much higher deduplication ratio than chunk-level deduplication [37]. We use the term fine-grained deduplication, though some previous literature uses the term delta compression to refer to this entire process.

However, fine-grained deduplication's performance is usually much worse than that of chunk-level deduplication because of further reducing data locality. Chunk-level deduplication usually suffers from the poor locality of deduplicated data, which has been mentioned in several previous works [12, 23, 58]. For example, when deduplicating a workload, we only store unique chunks and "share" chunks that appear in stored workloads as duplicates. Because chunks are stored in chronological order, this kind of "sharing" results in duplicate chunks and other unique chunks of this workload being scattered across the storage media, which leads to poor performance when restoring this workload. This problem is aggravated by fine-grained deduplication. It is because fine-grained deduplication introduces delta compression to exploit more compressibility among workloads, so workloads "share"

more data, decreasing locality, increasing I/O overheads, and leading to worse backup/restore performance.

Generally, different forms of the poor locality caused by delta compression impact backup and restore workflows. In the backup workflow, reading base chunks for delta encoding suffers from poor locality of base chunks (denoted by **Reading Base Issue**). Specifically, this issue is related to local compression, since consecutive chunks are compressed and must be decompressed together, which makes the compression unit become the I/O unit [6, 25, 44]. Thus, we have to read a compression unit even when only one or a few base chunks are needed, which leads to huge I/O amplification. In the restore workflow, the **Fragmentation Issue** [12, 23, 53] (that also exists in chunk-level deduplication) is exacerbated by the more complex dependencies in fine-grained deduplicated data. It is caused by a new kind of reference relationship between delta and base chunks, and this new kind of reference relationship further breaks spatial locality in fine-grained deduplicated data. Meanwhile, additional reference relationships between delta and base chunks also lead to a poor temporal locality in fine-grained deduplicated data. During delta decoding, base chunks and delta chunks must both be read, unlike restoring deduplicated data that only requires a single I/O read for a needed chunk, which makes the restore workflow repeatedly access containers to gather delta-base pairs (denoted by the **Repeatedly Accessing Issue**).

In this paper, we aim to improve these locality issues based on several observations and techniques.

For the *Reading Base Issue*, we apply a backup-workflow-oriented delta selector to improve the efficiency of reading base chunks in the backup workflow. It is based on an observation that most base chunks are located in a few containers (e.g., 64.1% containers only include 8.31% of the base chunks when running a backup workflow in a studied dataset). According to this observation, our delta selector skips delta compression when base chunks are located in those “base-sparse containers”. Without reading these “inefficient” containers, the efficiency of reading base chunks will be improved.

For the *Fragmentation Issue*, we propose a delta-friendly data layout, which covers the two-level reference relationships in fine-grained deduplicated data: the chunks–workloads reference relationship (also exists in chunk-level deduplication) and the additional delta–base reference relationship (caused by delta compression). The delta-friendly data layout handles the new dependencies and improves the spatial locality in fine-grained deduplicated data.

For the *Repeatedly Accessing Issue*, we observe the existence of “Always-Forward-Reference” traversing. It is a special path to traverse restore-involved containers, in which delta chunks always appear before their base chunks. By using this feature and exploiting the asymmetry of the I/O characteristics of storage media, we design a delta prewriting mechanism to deal with the poor temporal locality in deduplicated data, which first prewrites delta chunks to their

location in the to-be-restored workload and then reloads them for decoding when later accessing their base chunks.

We propose MeGA, a fine-grained deduplication framework, by using the above techniques to address the *Reading Base Issue*, *Fragmentation Issue*, and *Repeatedly Accessing Issue*. As shown in Fig. 1, MeGA achieves performance close to chunk-level deduplication while preserving fine-grained deduplication’s significant deduplication ratio advantage. The contributions of this paper are threefold:

- We analyzed several forms of poor locality caused by fine-grained deduplication, which leads to additional I/O overhead and poor backup/restore performance.
- We proposed techniques (i.e., the backup-workflow-oriented delta selector, the delta-friendly data layout, the “Always-Forward-Reference” traversing, and delta prewriting) to deal with these different issues caused by the poor locality.
- We proposed MeGA with these techniques to achieve performance close to chunk-level deduplication while preserving fine-grained deduplication’s significant deduplication ratio advantage. Especially, compared with the traditional greedy approach [53], MeGA achieves a 4.47–34.45 \times higher backup performance and a 30–105 \times higher restore performance, while maintaining a very high deduplication ratio.

2 Background and Related Works

2.1 Fine-grained Deduplication

Fine-grained deduplication [10, 15, 37, 38, 44, 51] could achieve a much higher deduplication ratio than deduplication alone [9, 11, 17, 19, 21, 32–34, 39]. It focuses on redundancies not only between duplicate chunks but also between similar but non-identical chunks, and finally achieves sub-chunk-level detection as well as byte/string-level elimination.

However, fine-grained deduplication achieves a higher deduplication ratio while introducing additional computation and I/O overhead when applying delta compression between similar chunks. To address these challenges, many previous works have been proposed, and the additional computation overhead has been hugely reduced. For example, Zhang et al. [52] and Zou et al. [57] proposed much faster sketch methods by exploiting the locality in backup streams and content-based sampling, respectively. MacDonald [26] proposed Xdelta for fast delta encoding. Xia et al. [48, 49] and Tan et al. [41] presented chunking-inspired methods to further improve delta encoding/decoding speeds. Zhang et al. [53] extended the rewriting techniques [12, 23] from chunk-level deduplication to fine-grained deduplication to reduce the additional I/O overhead only in fine-grained deduplication’s restore workflow.

Fine-grained deduplication has been employed in many other works. Xu et al. [50] introduced fine-grained deduplication for databases to reduce storage cost. Jain et al. [16] applied the idea of fine-grained deduplication in replica syn-

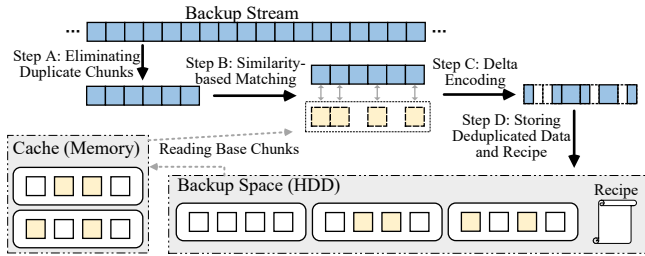


Figure 2: The backup workflow of fine-grained deduplication.

chronization. Pucha et al. [35], Mogul et al. [30], and Zhou et al. [55] designed a detection mechanism for p2p system, which finds both identical and similar sources to accelerate downloads.

Fig. 2 shows a standard workflow for fine-grained deduplication: ① Split backup streams into chunks and calculate a fingerprint for each chunk. ② Check and eliminate duplicate chunks using the fingerprint index. ③ Calculate each unique chunk’s sketches. Super Feature [5, 22, 24, 37] is a typical kind of sketch. It first generates multiple local-sensitive hashes with rolling hashes and linear transformations, and then packs these local-sensitive hashes together into fewer Super Features to detect highly similar chunks. ④ Find similar candidates for unique chunks using a sketch index or cache. ⑤ If a similar candidate exists, read it as a base chunk, and delta encode the incoming chunk relative to the base, often generating a much smaller delta chunk. ⑥ All deduplicated chunks are stored in containers in order, and then each container will be compressed. ⑦ Generate a recipe for a backup stream by recording fingerprints of all needed chunks, including indirectly referenced base chunks.

2.2 Backup Workloads

In backup storage systems [29], workloads usually are a series of backups (i.e., successive snapshots of the primary data), and consecutive backups are usually similar, which has been reported and exploited in many existing studies [13, 47, 52]. Thus, due to the highly redundant nature of the data, backup storage often leverages data deduplication to greatly reduce the size of backups and save hardware costs.

Deduplicated data (i.e., chunks) are usually locally compressed and stored in immutable and fixed-size containers (e.g., 4MB). Containers are compatible with striping across multiple drives in a RAID configuration, and writing in large units achieves the maximum sequential throughput [23].

3 Observation and Motivation

3.1 Challenges

Fine-grained deduplication obtains a higher deduplication ratio than chunk-level deduplication with much worse backup/restore performance, but further fragments data locality. As mentioned in several previous works [12, 23, 58], chunk-level deduplication usually suffers from poor locality because chunks from a workload that are logically consecu-

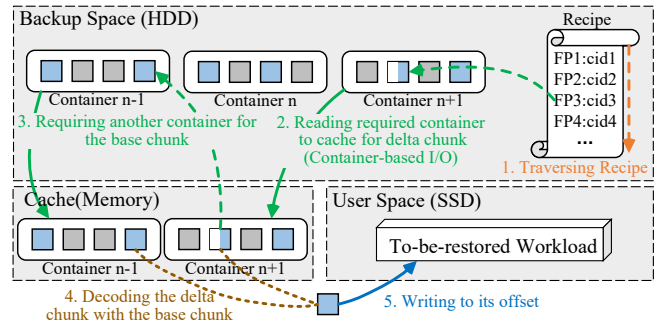


Figure 3: Restoring a delta chunk in the restore workflow of fine-grained deduplication.

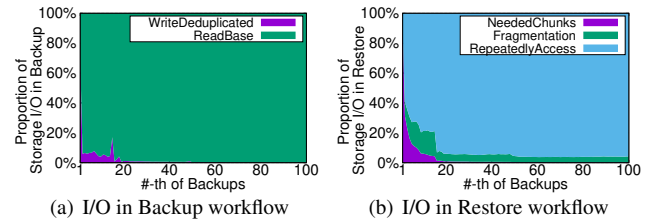


Figure 4: I/O overheads in backup and restore workflow.

tive may refer to previously written chunks scattered across the disks. However, fine-grained deduplication has more serious locality issues. Specifically, fine-grained deduplication eliminates redundancies among similar chunks by creating more references to previously written chunks, which increases fragmentation. Meanwhile, this observation also means that as more space is saved, locality becomes worse. Poor locality harms both backup and restore performance.

Fig. 2 demonstrates the poor locality involved in reading base chunks for fine-grained deduplication. Specifically, this issue is related to the local compression, since consecutive chunks must be decompressed entirely according to the compression unit (that also becomes the I/O unit) [6, 25, 44], which could be containers or compression regions (i.e., containers’ sub-unit). Therefore, reading a compression unit when only one or a few base chunks are needed leads to I/O amplification. Generally, a larger I/O unit (e.g., containers) may cause a larger I/O amplification, but it also could opportunistically prefetch more base chunks and reduce costly random accesses on HDDs (due to locality of backup stream [38, 52]). Even with a smaller I/O unit (e.g., 128KB container regions), reading bases remains a bottleneck [38]. Though it may cause less I/O amplification, reading some base chunks having locality with a small I/O unit can be disrupted by write tasks and result in more random seeks, because the backup workflow of fine-grained deduplication mixes reads and writes (i.e., reading base chunks and writing deduplicated data). Thus, we learn **Challenge 1**: Poor locality in the backup workflow causes inefficient I/O when reading base chunks.

In the restore workflow (like Fig. 3), there are two challenges. The first challenge in the restore workflow is the fragmentation problem, which is caused by poor spatial locality in

deduplicated data. It also exists in chunk-level deduplication, but it becomes more serious in fine-grained deduplication. It is because fine-grained deduplication allows workloads to share more similar chunks, but it also produces more references to previously written chunks. Therefore, fine-grained deduplication introduces another kind of reference relationship (i.e., between the base and delta chunks) and no longer only has one kind of reference relationship (i.e., between chunks and workloads). This makes the fragmentation problem more complex since the dependencies of each workload are distributed more widely. Thus, there exists **Challenge 2: Delta-base relationships lead to more complex fragmentation problems than deduplication alone.** The restore workflow also has **Challenge 3: Delta-base dependencies cause poor temporal locality during delta decoding and causes repeated container reads.** Without fine-grained deduplication, individual chunks can be read as needed to restore a file, but for fine-grained deduplication, base chunks and delta chunks must both be read. When chunks in a container are used (for unique or base chunks) across long time intervals, the restore workflow needs to alternately and repeatedly access containers to gather delta-base pairs for delta decoding.

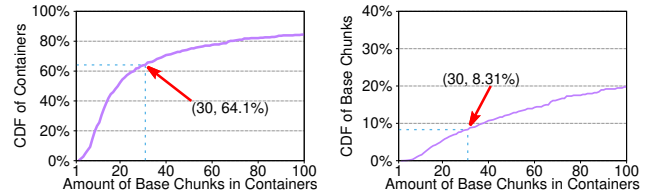
Finally, Fig. 4 suggests the seriousness of these challenges. It studies the I/O overheads of a basic fine-grained deduplication system with container I/O when backing up and restoring backup workloads from a WEB dataset (detailed in §5.1), which consists of 100 snapshots of a website. “WriteDeduplicatedData” means I/O for writing deduplicated data in the backup workflow, and “NeededChunks” means I/O for reading needed chunks. “ReadBase”, “Fragmentation”, and “RepeatedlyAccess” map to the above three challenges, respectively. We learn that these three challenges cause huge I/O overheads, and even “WriteDeduplicated” and “NeededChunks” only take about 0.3% and 1.12% of the total I/O in backup and restore workflows.

3.2 Selective Delta Compression

As Challenge 1 mentioned, poor locality in reading base chunks causes large I/O overheads in the backup workflow.

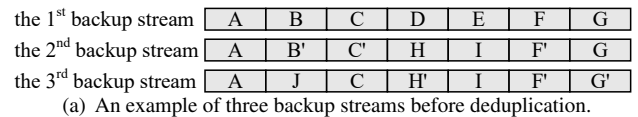
To this end, we studied datasets and observed that base chunks are not distributed evenly. For example, Fig. 5 gives the distribution of base chunks when backing up the 100th backup in the WEB dataset. Fig. 5(a) suggests that 64.1% of containers include fewer than 30 base chunks, and Fig. 5(b) demonstrates that these containers only hold 8.31% of the total base chunks. We call these containers “base-sparse containers”. Though there are only a few base chunks in these base-sparse containers, when requiring base chunks in one of them, we have to load the whole container from the disk, which causes a significant read amplification.

Thus, these observations motivate us to design a **backup-workflow-oriented delta selector**, which skips delta compression whose base chunks are located in “base-sparse containers” to avoid reading these “inefficient” containers. Thus,

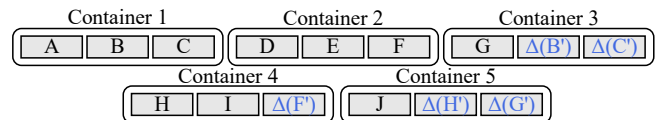


(a) 64.1% of containers contain only ~30 base chunks. (b) These 64.1% containers only include 8.31% of the total base chunks.

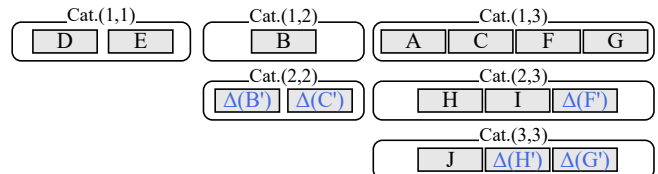
Figure 5: Base chunks are not distributed evenly.



(a) An example of three backup streams before deduplication.



(b) The order-based data layout after fine-grained deduplicated.



(c) The delta-friendly data layout for fine-grained deduplication.

Figure 6: An example of the order-based data layout versus the delta-friendly data layout.

it could reduce the I/O overheads in the backup workflow, and finally greatly improve the backup speed in fine-grained deduplication, which will be evaluated in §5.2.

3.3 Delta-friendly Data Layout

For Challenge 2, we use the example in Fig. 6 to discuss the fragmentation problem in fine-grained deduplicated data. Fig. 6(a) lists three backup streams, and Fig. 6(b) suggests the order-based data layout after fine-grained deduplicating these three backup streams. The order-based data layout allocates chunks in containers according to their written order and is widely used in previous works [12, 23, 37, 52]. When restoring a backup, the needed and unneeded chunks are always mixed in this data layout. Consider Container 2 in Fig. 6(b) for example: when restoring the 3rd backup, chunk *F* is needed while chunks *D* and *E* are unneeded, but all of them will be read as a whole container due to container I/O, which causes extra I/O overheads.

Rewriting-like defragmentation approaches could be extended to fine-grained deduplication to alleviate the fragmentation problem [53]. Their mechanisms can be summarised as skipping deduplicating chunks already in sparse containers, but this cannot stop the locality of deduplicated data becoming increasingly poorer as the number of backups increases, which thus makes the restore speed continually decrease [13, 23, 53].

MFDedup [58] introduces a lifecycle-based data layout and eliminates the fragmentation problem in chunk-level deduplication. The lifecycle-based data layout classifies chunks into categories according to whether they are always referenced by the same set of consecutive backup workloads (i.e., **lifecycles**), and stores chunks in the same category together. Lifecycle-based classification of chunks ensures whichever backup workload is to be restored, chunks in any categories are always either all needed together or all not needed together. Thus, reading needed chunks in the unit of categories will never cause unneeded chunks to be read. Generally, MFDedup only considers **one-level** simple reference relationships (between chunks and backup workloads), which is the only type of reference relationship in chunk-level deduplication.

However, directly applying this lifecycle-based data layout to fine-grained deduplication is not feasible since fine-grained deduplication introduces an additional kind of reference relationship between delta and base chunks and causes new fragmentation. In the 2nd backup stream, there are **two-level** reference relationships:

- Between workloads and chunks.
i.e., the 2nd backup stream $\Leftrightarrow \{A, B', C', H, I, F', G\}$
- Between base chunks and delta chunks.
i.e., $B \Leftrightarrow \Delta(B)$; $C \Leftrightarrow \Delta(C)$; $F \Leftrightarrow \Delta(F)$

Therefore, we need a new data layout that considers both kinds of reference relationships to eliminate the fragmentation problem in fine-grained deduplicated data.

We first need a new way to describe chunks' lifecycles with the additional introduced reference relationship's impacts. Here we define the **Necessary Chunks** (denoted by **NC**) of a backup workload as the combination of its directly referenced chunks (i.e., the 1st level) and its indirectly referenced chunks (i.e., the 2nd level). Accordingly, we redefine a chunk's lifecycle in fine-grained deduplication as which backup workloads' NCs refer to this chunk, which could cover the two-level reference relationships. In Fig. 6, we can list the NCs for the three backups:

- NC_Backup1: A, B, C, D, E, F, G
- NC_Backup2: A, B, $\Delta(B')$, C, $\Delta(C')$, H, I, F, $\Delta(F')$, G
- NC_Backup3: A, J, C, H, $\Delta(H')$, I, F, $\Delta(F')$, G, $\Delta(G')$

In this example, the lifecycle of chunk *G* is from *NC_Backup1* to *NC_Backup3*, since *G* is used as a unique chunk for *NC_Backup1* & *NC_Backup2* and then as a base for *NC_Backup3*.

After that, we could build a **delta-friendly data layout** by integrating the second level of reference relationship into the lifecycle management as well. As shown in Fig. 6(c), the delta-friendly data layout consists of categories, which includes several chunks. To clearly present them, we use **Cat.(X,Y)** to indicate the category, which includes all chunks whose lifecycles are only from *NC_BackupX* to *NC_BackupY*. All deduplicated data are classified and sequentially stored in categories according to their lifecycles, which hugely benefits the restore workflow. In this example, *NC_Backup1* is composed of

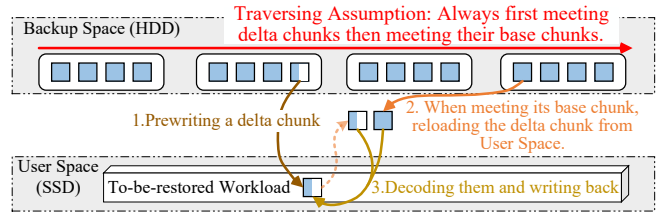


Figure 7: The delta prewriting mechanism. Here the half shaded chunk is a delta chunk.

Cat.(1,1), *Cat.(1,2)* and *Cat.(1,3)*; *NC_Backup2* is composed of *Cat.(1,2)*, *Cat.(2,2)*, *Cat.(1,3)* and *Cat.(2,3)*; *NC_Backup3* is composed of *Cat.(1,3)*, *Cat.(2,3)* and *Cat.(3,3)*. When restoring any of these three backups, we can select categories according to the above lists, and all chunks in selected categories are all needed. In this way, the restore workflow never needs to read any unneeded chunks, and the *Fragmentation Issue* in Challenge 2 could be eliminated.

To simplify the implementation of the delta-friendly data layout, we only deduplicate redundancies between adjacent backups to ensure that chunks' lifecycles are always consecutive (composed of successive backup streams' *Necessary Chunks*), similar to the approach in MFDedup [58]. This strategy may reduce the total deduplication ratio, but it will not be significant according to several previous works [38, 44, 58], which will be also further studied in §5.4.

3.4 Forward Reference and Delta Prewriting

For Challenge 3, we design a delta prewriting mechanism. It relies on two things: ① The storage media's I/O characteristics between User Space and Backup Space are asymmetric. Backup Space usually uses HDDs as storage media due to its lower price, while User Space usually uses SSDs or NVMs since better I/O performance is essential for business [58]. ② When performing a restore, delta-encoded chunks are always accessed before their base chunks, which we call "**Forward Reference.**"

Fig. 7 shows the basic idea of the delta prewriting mechanism. For each delta chunk, the prewriting mechanism will prewrite it to the offset where it should be after delta decoding in the to-be-restored backup workload (in User Space). And then, when meeting its base chunk later, the prewriting mechanism will read the delta chunk from the prewritten position, decode the delta chunk with the base, and finally write back the decoded chunk to its offset. Through this mechanism, we ensure that when restoring, all restore-involved containers only need to be read only once, which hugely reduces the I/O overheads on Backup Space.

The next issue is how to make the assumption always hold. By studying the data layout proposed in §3.3, we find it is possible to design a special path for traversing restore-involved containers when restoring, in which delta chunks always appear in front of their base chunks. We call it "**Always-Forward-Reference**" **traversing** (shortened to

AFR traversing), whose details will be introduced in §4.4.

Due to improved spatial locality (delta-friendly data layout in §3.3) and temporal locality (the AFR traversing and delta prewriting in §3.4) in deduplicated data, the I/O overheads in the restore workflow are hugely reduced. Meanwhile, there exists only sequential I/O to the Backup Space when restoring, which is optimized for HDDs. Finally, the restore speed could be greatly improved, which we evaluate in §5.3.

4 Design and Implementation

4.1 General Description

The overall framework of MeGA is shown in Fig. 8. In general, ① For the backup workflow, MeGA first runs *Chunk-level Deduplication* to remove duplicate chunks according to *Local-based FP Index*, and then, it finds similar matches for unique chunks according to *Local-based Sketch Index* and selectively applies delta compression using *Delta Selector*. ② For the storage organization, MeGA stores and manages the deduplicated and delta compressed data in the *Delta-Friendly Data Layout*. ③ For the restore workflow, MeGA generates an *Offset Hash Table* according to the recipe of a to-be-restored workload; then, MeGA accesses all restore-involved containers with *AFR Traversing* and *Delta Prewriting*.

Specifically, there are several modules in MeGA:

- *Local-based FP Index* and *Local-based Sketch Index* maintain fingerprints and sketches of each backup workload’s chunks in separate hash tables per backup. They only retain the current and last backup’s tables (similar to some previous works [44, 58]), because MeGA only deduplicates a backup within itself and the previous backup (mentioned in §3.3).
- *Chunk-level Deduplication* first splits the backup stream into chunks with Content-Defined Chunking [31, 46] and then calculates a fingerprint (i.e., SHA1 digest) for each chunk. After that, it detects and eliminates identical chunks with a Local-based FP index.
- *Delta Selector* first generates sketches with the resemblance detection approaches [4, 5, 37, 52, 57] for unique chunks and identifies similar candidates according to the Local-based Sketch index for further delta compression. Then, it delta-encodes chunks unless the referenced bases are in base-sparse containers.
- *Base Cache* holds cached containers to provide base chunks for delta compression in the backup workflow.
- *Delta-Friendly data layout* manages fine-grained deduplicated chunks according to their lifecycles, reflecting which backup workloads require these chunks. As a result, the delta-friendly data layout promises to eliminate the fragmentation problem in fine-grained deduplicated data and reduce I/O overheads in the restore workflow.
- *AFR Traversing* applies “Always-Forward-Reference” traversing on fine-grained deduplicated data in a restore workflow, which guarantees that delta chunks are always accessed before their base chunks and provides the pre-

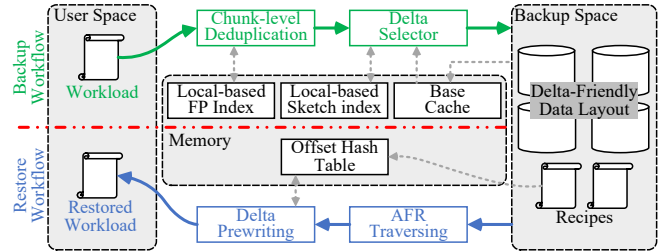


Figure 8: An overview of MeGA framework.

condition for Delta Prewriting.

- *Delta Prewriting* transfers the random operations from Backup Space to User Space, exploiting the asymmetry of storage media characteristics between the two spaces. This also avoids repeatedly accessing containers when restoring files.
- *Offset Hash Table* is built according to a to-be-restored backup workload’s recipe and provides offsets of chunks (three kinds of chunks: unique, base, and delta) in the to-be-restored backup workload.

Details of each workflow using our proposed key techniques will be introduced in the following §4.2–§4.4.

4.2 Backup Workflow

The backup workflow runs *Chunk-level Deduplication* and *Delta Selector* to eliminate duplicate chunks and redundancies among similar chunks, respectively.

Chunk-level Deduplication. The chunk-level deduplication step splits the backup stream into chunks with Content-Defined Chunking [31, 46] and then calculates a fingerprint (i.e., SHA1 digest) for each chunk. After that, MeGA detects and removes duplicate chunks according to the Local-based FP Index, as we introduced in §4.1.

Delta Selector. Then, the backup workflow runs *Delta Selector* with the following steps. ① *Delta Selector* first combines several successive chunks (from *Chunk-level Deduplication*) into fix-sized segments (e.g., 20MB). ② In each segment, *Delta Selector* generates sketches (i.e., Super Features [22]) for each (unique) chunk, and then tries to find for each chunk a similar chunk as its base chunk with the Local-based Sketch Index. ③ For chunks that have a potential base chunk, *Delta Selector* records their base chunk’s container ID in a ‘selector table’, which counts the times each container is referenced for base chunks within a segment. ④ Then, *Delta Selector* observes which containers are rarely referenced (with a **threshold**) in the ‘selector table’ and considers these containers as ‘sparse-base containers’, which are inefficient to read for base chunks. ⑤ Finally, for chunks having a similar chunk that is not in sparse-base containers, *Delta Selector* will run delta compression to calculate and store their differences (i.e., delta chunk) for saving space; For the remaining chunks, they will be directly stored as unique chunks. Base chunks in delta compression are acquired from the base cache, and if a cache miss occurs, the base cache will read related containers

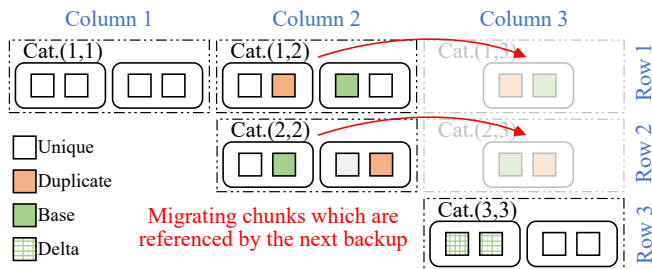


Figure 9: An example of maintaining data layout after storing the 3rd backup. Chunks, which are duplicate to the 3rd backup and referenced as base chunks in the 3rd backup, will be migrated to new categories. *Cat.(1,3)* and *Cat.(2,3)* do not exist before migrations.

from disks and add them to the cache.

As a result, Delta Selector could improve the efficiency of reading base chunks and then accelerate the backup workflow. Next, we will introduce how to store these deduplicated data.

4.3 Maintaining Delta-Friendly Data Layout

In this subsection, we will introduce how to locate the deduplicated data in the delta-friendly data layout. There are two steps: ① Store the incoming deduplicated data of a new backup in the delta-friendly data layout. ② Process the incoming and previous backups’ deduplicated data to ensure each chunk’s location is consistent with the principle of our delta-friendly data layout.

Storing New Fine-grained Deduplicated Data and Data Organization. For storing fine-grained deduplicated data, we first consider their lifecycles. After running the backup workflow (introduced in §4.2), the fine-grained deduplicated data consists of the latest backup workload’s unique and delta chunks. Since these chunks are only referenced by the latest backup, they should have the same lifecycle, and their lifecycle should be different from previously stored chunks.

Then, considering the definition of the lifecycle and the naming style of categories (**shorten to Cat.**) introduced in §3.3, these chunks (assuming they are from the n^{th} Backup) should be classified into a new category *Cat.(n,n)*.

Considering the sizes of categories are usually variable, we design a two-level storage organization: fix-sized Containers (e.g., 4MB) and variable-sized Categories. Containers directly hold chunks, and categories hold containers whose chunks have the same lifecycle. For example, *Cat.(1,2)* could include one or several containers, and each container holds chunks whose lifecycle is from *NC_Backup1* to *NC_Backup2*.

Data Migration. After storing fine-grained deduplicated data of the latest backup workload, we should consider updating the data layout to handle the issue that some chunks’ lifecycles are changed. In general, storing a new backup in the delta-friendly data layout only changes the lifecycles of its adjacent backups’ chunks, because MeGA only allows adjacent backups to share common chunks (i.e., MeGA deduplicates a backup within itself and its previous backup). Therefore,

these shared chunks’ lifecycles should be extended to the latest backup. Thus, we need to migrate these shared chunks into new categories to match their updated lifecycles, and we call these migrations the maintenance workflow.

An example of the maintenance workflow is shown in Fig. 9. It shows a situation that the 1st and 2nd backups have been stored in the data layout, and the 3rd backup is the latest one, whose fine-grained deduplicated data have been stored in *Cat.(3,3)*, as discussed earlier in this subsection. At this time, some chunks located in *Cat.(1,2)* and *Cat.(2,2)* are referenced by the 3rd backup (as duplicate or base chunks). Thus, these chunks’ lifecycles newly include *NC_Backup3* and they should be migrated into new categories. In this example, chunks in *Cat.(1,2)* and *Cat.(2,2)* will be traversed, and duplicate/base chunks will be migrated into new categories *Cat.(1,3)* and *Cat.(2,3)*, respectively.

Note that the **maintenance workflow (i.e., data migration) only works on related categories and does not involve all categories.** As the example in Fig. 9 shows, a maintenance workflow after storing the 3rd backup only impacts Column 2. Similarly, the maintenance workflow always runs on one column, and its overhead is also limited (will be studied in §5.6). With support of the maintenance workflow, the delta-friendly data layout is preserved, which benefits restore performance.

Features in Migration. There exist two interesting features when the maintenance workflow involves delta and base chunks. For clarity, here we say *Cat.(X,Y)* is in Row X, and Column Y, as shown in Fig. 9.

Feature 1: base chunks are always in the same or an earlier Row than their delta chunks. It could be easily explained by the example in Fig. 9. For delta chunks of the 3rd backup (must be in *Cat.(3,3)*), their base chunks can only be from two sources: ① from the 3rd backup itself. In this case, the bases are also in *Cat.(3,3)*, the same Row as the delta. ② from the 2nd backup. In this case, the bases must be migrated into *Cat.(1,3)* or *Cat.(2,3)*, the earlier Row than the delta.

Feature 2: base chunks are always in the same or a later Column than their delta chunks. Here we also take Fig. 9 as an example: When the duplicate chunks in Fig. 9 contain base or delta chunks, there are two cases: ① If a delta chunk is a duplicate of another delta chunk (i.e., its “original” chunk is a duplicate of delta-encoded chunk) to the 3rd backup and should be migrated, its base must be also migrated as the delta’s dependency, since both of them should be included in *NC_Backup3*. Therefore, in this case, they will be migrated into the same Column (Column 3). ② If a base chunk is duplicate (i.e., is itself a duplicate) to the 3rd backup and should be migrated, its delta will not be migrated, since the 3rd backup does not require this delta chunk. In this case, the base (migrated to Column 3) will be in a later Column than the delta (left in Column 2).

These two features will help to achieve “Always-Forward-Reference” traversing, which will be further used in §4.4.

Table 1: Possible category locations of the corresponding base chunks for the delta chunks in the 2nd backup.

Delta Chunks' Positions	Corresponding Base Chunks' Possible Positions
Cat.(1,2) ⇒	Cat.(1,2), Cat.(1,3)
Cat.(2,2) ⇒	Cat.(1,2), Cat.(2,2), Cat.(1,3), Cat.(2,3)
Cat.(1,3) ⇒	Cat.(1,3)
Cat.(2,3) ⇒	Cat.(1,3), Cat.(2,3)

4.4 Restore Workflow

As introduced in §4.1, the restore workflow of MeGA relies on AFR traversing and Delta Prewriting. In the beginning, the restore workflow needs to determine which containers are needed for restoring the required backup workload.

Identifying All Required Containers. All the required containers could be simply calculated in a delta-friendly data layout. For example, there are n backup workloads stored, and we want to restore a backup B_k . According to the naming style of categories (mentioned in §3.3), all categories whose lifecycles include NC_Backup_k are required, and they are $\bigcup_{j=k}^n \bigcup_{i=1}^j Cat.(i, j)$, where $1 \leq i \leq k \leq j \leq n$. For example, when restoring the 2nd backup in Fig. 9, $Cat.(1,2)$, $Cat.(2,2)$, $Cat.(1,3)$, $Cat.(2,3)$ are required.

Then all containers in these categories are the restore-required ones. Benefiting from the delta-friendly data layout, all chunks in these containers are exactly what we need, which avoids reading unneeded chunks when restoring. Next, we present how to traverse them for restoring a workload.

AFR Traversing. As mentioned in §4.1, AFR traversing promises that when traversing the restore-involved containers, delta chunks always appear in front of their base chunks. For the example in Fig. 6(c), when restoring the 2nd backup, restore-involved categories are $Cat.(1,2)$, $Cat.(2,2)$, $Cat.(1,3)$ and $Cat.(2,3)$ (according to “Identifying All Required Containers”). In this case, we can achieve AFR traversing with the following order: $Cat.(2,2) \Rightarrow Cat.(1,2) \Rightarrow Cat.(2,3) \Rightarrow Cat.(1,3)$, in which we always meet the delta chunks before their base chunks.

Next, we explore how and why MeGA can achieve AFR traversing, also with the example of restoring the 2nd backup in Fig. 6(c). Consider the two key Features about the relative positional relationship (i.e., the located categories' Rows and Columns) between the delta and base chunks (learned from §4.3). We can get Table 1, listing all possible positions (i.e., located categories) of delta and based chunks of the 2nd backup. To achieve AFR traversing (accessing delta chunks and then their bases), $Cat.(2,2)$ must be first accessed, which is because the base chunks of $Cat.(2,2)$'s delta chunks could be in all four categories as shown in Table 1. With similar analysis, we could finally get the previous example path: $Cat.(2,2) \Rightarrow Cat.(1,2) \Rightarrow Cat.(2,3) \Rightarrow Cat.(1,3)$. Additionally, AFR traversing should go through chunks and also containers of each category in reverse order in case there are delta and base chunks in the same category or container, since the delta

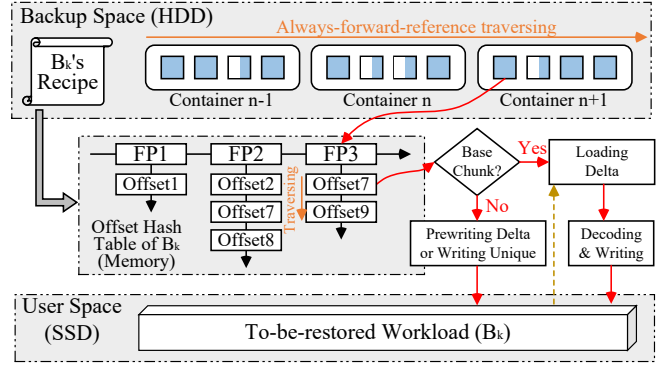


Figure 10: An example of the restore workflow.

must be generated and then appear after its base in the backup workflow.

To this end, we can summarize **three rules to achieve AFR traversing** on our delta-friendly data layout in general cases:

- Between columns, access columns in positive order. This is deduced from Feature 2 (in Section 4.3).
- In the same column, access categories in reverse order. This follows from Feature 1 (in Section 4.3).
- In a category, access containers in each category and chunks in each container in reverse order. This is because delta chunks can only reference earlier chunks by design.

Delta Prewriting. As shown in Fig. 10, *Delta Prewriting* requires an *Offset Hash Table*, which is generated according to the to-be-restored backup's recipe. The Offset Hash Table records key/value pairs: **each chunk's offset** (in the to-be-restored backup) and **whether it is a base chunk** (i.e., <offset, base tag>). For unique chunks in the recipe, we only insert its offset into its FP's entry list and tag this record as not a base (e.g., <offsetUniqueK, false>). For a delta chunk in the recipe, we first process it as a unique chunk (e.g., insert <offsetDeltaN, false> in its FP's entry list) and then additionally insert a record into the entry list of its base's FP and tag this record as a base chunk (e.g., <offsetDeltaN, true>).

Then, we apply AFR traversing on restore-involved containers. For each chunk, we acquire its entry list according to its fingerprint. We check each record in the entry list: If it is not a base chunk record, we directly write the chunk (it may be a delta or unique chunk) to the offset in the record; If it is a base chunk, we read the delta chunk from the offset in the record (the delta should already be written before), decode the delta chunk with the base chunk, and then write back the decoded chunk to the offset in the record.

Finally, MeGA could achieve a much higher restore speed with the benefits of the delta-friendly data layout, AFR traversing and delta prewriting, since it no longer reads unneeded chunks and repeatedly accesses restore-involved containers.

4.5 Discussion

In this subsection, we discuss several features and issues.

Deletion. Different from the order-based data layout, the delta-friendly data layout supports direct deletions without GC. Because MeGA allows workloads to share chunks as duplicate chunks or base chunks, deleting the n^{th} backup only needs to remove its unique chunks. According to the category naming rule, $Cat.(n,n)$ only contains chunks unique to backup n . Thus, deleting the n^{th} backup could be achieved by directly removing this category, instead of the way in the order-based data layout, which first runs logical deletion and later runs garbage collection to reclaim storage space [3, 9, 14].

Delta Prewriting. This mechanism introduces additional I/O on User Space, including prewriting and reading delta chunks. Our observations suggest these issues cause about 5%-10% additional I/O overheads on User Space. Moreover, since delta chunks are usually much smaller than unique ones, we could also introduce a delta cache in memory and prewrite delta chunks into the cache as an alternative solution.

Memory Overhead. Since the size of the base chunk cache (in a backup workflow) is configurable, the other memory overhead of MeGA is mainly from the local-based indexes (in backup workflow) and the offset hash table (in restore workflow). ① Instead of putting the whole index in memory, MeGA only maintains the index of the last two backups and thus costs less memory, which is similar to some previous work [44, 58]. Moreover, a stream-informed index [56] could also be applied to our local-based index to further reduce memory overhead. ② The overhead of the offset hash table is related to the number of chunks in a single backup. Some previous works [1, 43] suggest that the majority of single backups were 4–128GB, and for these cases, RAM usage for the Offset Hash Table could be 12.9–445.6MB, which is feasible for a server. To reduce this RAM usage for large backups, we could keep the offset hash table in an on-disk key-value store, but it would require indexing time.

Maintenance's (i.e., Migrations) Overheads. The Maintenance process in MeGA replaces Garbage Collection (GC) in previous works, and its overhead could be offset since both techniques are offline processes, which will be evaluated in §5.6. Through the Maintenance process, MeGA could achieve direct deletion (to immediately reclaim storage space) instead of logical deletion followed by GC. Besides, the Maintenance process also addresses two interesting issues [36]: knowing how much space will be freed after deletion and estimating the remaining logical space of a fine-grained deduplication system.

Incremental Backups. Although MeGA focuses on full backups (i.e., a full snapshot of primary storage), we present a plan to support incremental backups by generating “virtual” full backups.

When handling an incremental backups, we generate a “virtual” full backup according to the previous full backup’s recipe, and then process the “differences” included in the incremental backup. Specifically, ① non-modified (i.e., not listed in the incremental backup) parts of the “virtual” full

backup are duplicates, and we can directly copy corresponding records from the previous full backup’s recipe to the “virtual” full backup’s recipe; ② modified (i.e., listed in the incremental backup) parts have potentially new content, and we need to apply fine-grained deduplication. Chunk boundaries need to be recalculated due to the modified data regions, so we could combine the new data with their surrounding duplicate chunks to make up a local stream and run content-defined chunking on this stream to determine new chunks. Then, MeGA could process these chunks normally, and finally record these “modifications” in the “virtual” full backup’s recipe.

5 Evaluation

5.1 Configuration

We perform our experiments on a workstation running Ubuntu 18.04 with an Intel Core i7-8700 @ 3.2GHz CPU, 64GB memory, and a 7200rpm HDD. To better evaluate MeGA, the following five approaches are considered:

- **Greedy:** applying the greedy strategy for fine-grained deduplication, often evaluated as the baseline [44, 53].
- **FGD:** Fine-Grained Deduplication with the Capping rewriting technique [23], which skips some deduplication and delta compression whose duplicate chunks or base chunks are located in a few referenced containers. This is similar to a recent work called SDC [53].
- **CLD:** Chunk-Level Deduplication with the Capping rewriting technique [23], considered as a typical approach of chunk-level deduplication defragmentation.
- **MFD:** Chunk-level deduplication with the previous lifecycle-based data layout, which only deduplicates chunks between adjacent backups [58].

These approaches are implemented according to related papers, and they all follow these common configurations:

- Chunking backups uses FastCDC [46] with the minimum, average, and maximum chunk sizes set to 2KB, 8KB, and 64KB; SHA1 is used for chunk identification.
- Their resemblance detection generates 12 features and 3 super-features as sketches for each unique chunk, as suggested in previous works [22, 24, 37].
- Consecutive chunks are compressed together with ZSTD and stored in containers.
- The delta encoding stage uses Xdelta to calculate differences between unique chunks and its similar candidates, as configuration in previous works [52, 57].
- MeGA only requires a container cache in the backup workflow, and the other four non-trivial approaches require two container caches in both backup and restore workflows. The cache of each workflow totals 512MB. When loading base chunks into the cache, all approaches apply container I/O for fair comparison.

To focus on testing the performance of the deduplication storage side (i.e., running deduplication on HDD media), tested datasets are backed up from User Space (i.e., a

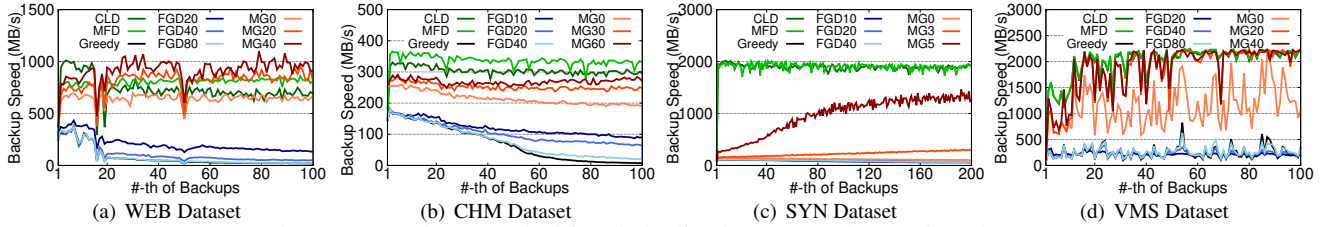


Figure 11: Backup speed of five deduplication approaches on four datasets.

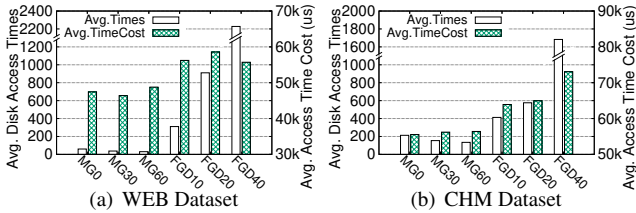


Figure 12: Disk access times and time cost in MeGA and FGD. We only show WEB and CHM due to the space limit.

Table 2: Four backup datasets used in evaluation.

Name	Original Size	Versions	Workload Descriptions
WEB	269 GB	100	Backups of website: news.sina.com, captured from Jun. to Sep. in 2016.
CHM	279 GB	100	Source codes of Chromium project from v82.0.4066 to v85.0.4165
SYN	1.38 TB	200	Synthetic backups by simulating file create/delete/modify operations [42]
VMS	1.55 TB	100	Backups of an Ubuntu 12.04 Virtual Machine

RamDisk) to Backup Space (i.e., a 7200rpm HDD) one by one while the restore runs in the reverse direction. For speed of backup and restore in our evaluation, we present the average results of five runs.

Four backup datasets are used for evaluation, as shown in Table 2. These datasets represent various typical backup workloads, including website snapshots, an open-source code project, virtual machine images, and a synthetic dataset. They have been used in several deduplication studies [8, 46, 53].

5.2 Backup Speed

The backup speed of five approaches are evaluated and shown in Fig. 11, and the results vary by 5.2% on average in multiple runs. FGD# and MG# represent FGD and MeGA with different parameters (the capping level in FGD and the delta selector threshold in MeGA). The capping level L indicates that when processing a backup stream segment, containers will be considered as sparse containers except for the L most referenced (for duplicate or base chunks). Chunks in the segment, whose duplicate or base chunks are in sparse containers, will be processed as unique chunks. A delta selector threshold T means that when processing a backup stream segment, containers, which are referenced for base chunks less than T times, will be considered as base-sparse containers. Delta compression in the segment, whose base chunks are in base-sparse containers, will be skipped. Considering that datasets

have different characteristics and require different parameters, we optimized the parameters for each dataset. The backup speed is calculated by $\frac{\text{The-Size-of-Backup-or-Restore-Workload}}{\text{Backup-or-Restore-Time-Cost}}$. Because deduplicated and compressed data is much smaller than their original size and writing to disk takes less time, the backup speed could exceed the disk speed.

With the benefits of the delta selector, MeGA outperforms other fine-grained deduplication approaches (i.e., Greedy and FGD). On the SYN dataset, MeGA reads increasingly fewer containers when processing later backups, which makes MeGA’s backup speed increase. VMS is a virtual machine dataset and its modification style (i.e., trending to change the same region in each backup) makes distribution of base chunks uneven, which makes MeGA’s performance jittery. Generally, MeGA achieves a 4.47–34.45 \times higher backup speed than Greedy.

Fig. 11 also suggests a stricter (smaller) capping level in FGD and a stricter (bigger) delta selector threshold in MeGA both accelerate backup speed, due to skipping some potential delta compression and the need to read more base chunks. Note that if delta selector threshold and capping level were strict enough, all delta compression would be skipped. Though the delta selector and the capping rewriting have similar mechanisms, their results are much different due to their different views on container utilization. The capping rewriting is restore-workflow-oriented and focuses on how many needed chunks (all kinds of chunks) are in containers. But the delta selector is backup-workflow-oriented, and only concerns how many base chunks are in containers.

Fig. 12 further studies why MeGA could achieve a much higher backup speed than FGD, which lists the disk access times for acquiring base chunks (within the unit of containers) and average access time cost when storing backup. On the one hand, MeGA has much lower disk access time due to skipping reading “inefficient” containers. On the other hand, MeGA has a lower average access time cost, since it only finds base chunks in adjacent backups and its accessed containers will be located closer. These two efforts ensures MeGA’s better performance.

Note that MeGA achieves similar results with chunk-level deduplication approaches (CLD and MFD) on most datasets. It is because the additional I/O and computation overhead both have been hugely limited by our delta selector and previous computation optimization works, respectively. Besides, SYN is a synthetic dataset and its modified parts are distributed ran-

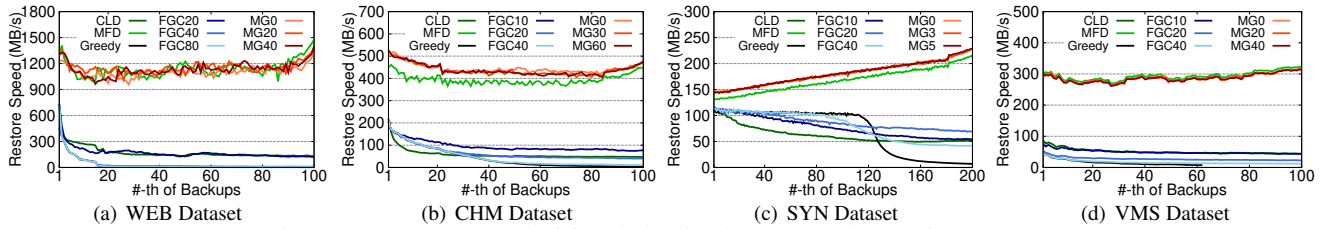


Figure 13: Restore speed of five deduplication approaches on four datasets.

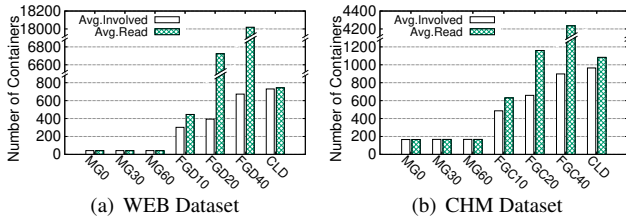


Figure 14: Number of restore-involved containers and actually read containers in MeGA, FGD and CLD. Only WEB and CHM are shown due to the space limit.

domly instead of having more typical locality, which makes MeGA slightly slower than CLD and MFD.

5.3 Restore Speed

Fig. 13 shows the restore speed of all five approaches, and the results vary by 2.6% on average in multiple runs. Among all approaches, MeGA consistently achieves a better restore performance than other approaches, which reflects MeGA’s restore techniques (i.e., the delta-friendly data layout, delta prewriting and AFR traversing). MeGA solves the *Fragmentation Issue* and *Repeatedly Accessing Issue* and improves the spatial and temporal locality in fine-grained deduplicated data. It also ensures that MeGA’s restore performance is more consistent, while FGD, CLD and Greedy all have a decreasing restore speed. Note that the local compression ratio increases when storing more backups on SYN, which makes the restore speed faster. Generally, MeGA achieves a 30–105× higher restore speed than Greedy.

Fig. 14 shows the number of restore-involved containers (i.e., containers with restore-required chunks) and containers read from disk during restore of MeGA, FGD, and CLD. These two metrics reflect the seriousness of the *Fragmentation Issue* and *Repeatedly Accessing Issue*, respectively. Compared with FGD and CLD, MeGA has lower results on both of the metrics due to applying our data layout and AFT traversing with delta prewriting. Consequently, MeGA achieves a much higher restore performance, as shown in Fig. 13.

5.4 Deduplication Ratio

Fig. 15 studies deduplication ratios of five approaches with different parameters mentioned in the above subsections. All three fine-grained deduplication approaches (i.e., Greedy, FGD, and MeGA) have higher deduplication ratios than chunk-level deduplication approaches (i.e., CLD and MFD), since they can exploit compressibility among similar chunks.

Greedy always achieves the highest deduplication ratio, and MeGA achieves similar results. For FGD and MeGA, a stricter capping level in FGD or threshold in MeGA will lower the deduplication ratio but lead to a better backup or restore speed, as reported in the above subsections.

MeGA’s advantage is relatively smaller on the VMS and SYN datasets. For VMS, its modification style (i.e., trending to change the same region in each backup) leads to fewer similar chunks, which limits the benefits of fine-grained deduplication, regardless of the approach. For SYN, its modifications are completely random because it is a synthetic dataset, and the locality of base chunks is not as strong as that of other datasets. Therefore, MeGA’s delta selector causes more reduction in the compression ratio.

Generally, MeGA preserves fine-grained deduplication’s significant advantage by achieving a 1.18–8.73× higher deduplication ratio than chunk-level approaches.

5.5 Overall Performance

The three metrics discussed above are of the most interest to users. Fig. 16 shows the overall performance with different parameters (used in Fig. 11 and 13) from the above section. It is obvious that MeGA significantly improves over other fine-grained deduplication approaches (i.e., FGD and Greedy) on both backup and restore speed while preserving the deduplication ratio advantage of fine-grained deduplication. It reflects the performance improvement that our proposed technology brings. MeGA’s advantage is relatively smaller on SYN and VMS datasets. As we mentioned in § 5.4, it is because VMS does not have many similar chunks, and SYN lacks natural locality, which is unfriendly for our delta selector.

5.6 I/O Overhead in Maintaining Data Layout

In this subsection, we evaluate I/O overheads of maintaining the delta-friendly data layout (shortened to "Maintenance") compared with traditional garbage collection (GC).

Our experiments are based on MeGA and FGD using the median parameters as in Fig. 11 and 13, and MeGA runs maintenance while FGD runs GC. For GC, a container liveness threshold is usually considered to make a tradeoff between more storage space cost and more GC overheads. Here we use three liveness thresholds: 0%, 25%, and 50%, mapping to toleration up to 0%, 25%, 50% invalid chunks in each container, respectively. In order to make the results among different datasets comparable, we measure their time costs. Both approaches retain the last 20 backups; thus GC would

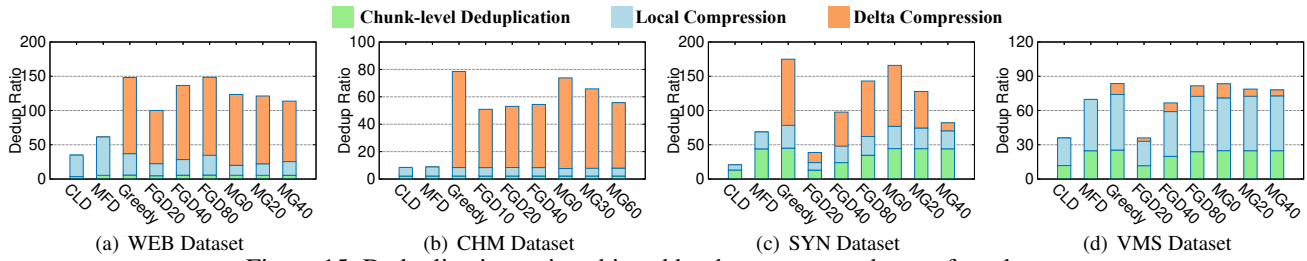


Figure 15: Deduplication ratio achieved by the ten approaches on four datasets.

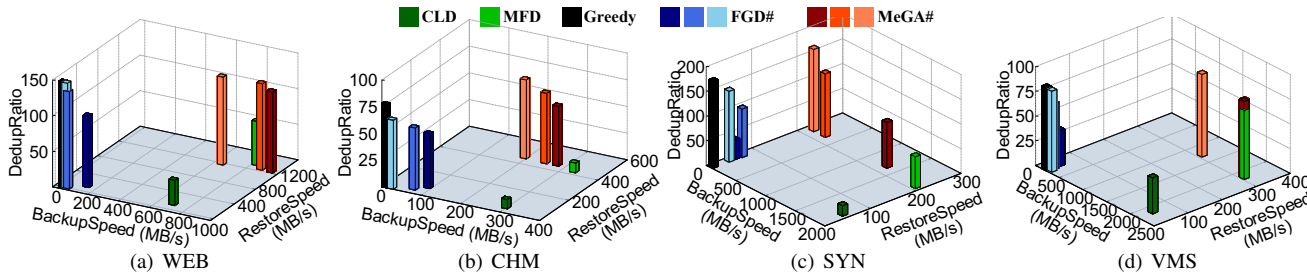


Figure 16: A general view of five approaches. MeGA and FGD use three different parameters as used in Fig. 11, 13 and 15. The results of backup and restore speed are from the average performance on each dataset’s last 10 backups.

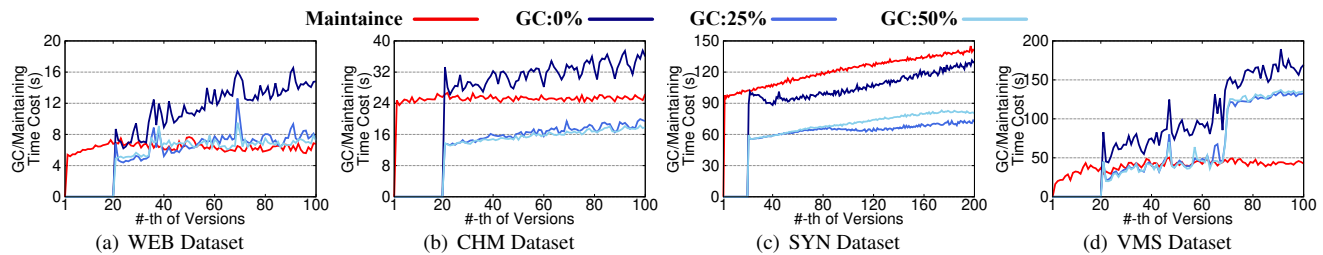


Figure 17: The delta-friendly data layout’s maintenance vs. the order-based data layout’s garbage collection.

not run for the first 20 backups, though the maintenance does.

Fig. 17 compares time cost of maintenance and GC. For GC, a bigger threshold does not always lead to a lower I/O overhead, since tolerating invalid chunks will make the next GC need to clean more containers, which causes additional I/Os. In general, maintenance and GC have similar I/O overheads, and compared with the best version of GC (“GC:25%”), maintenance costs about 0.32–1.92× the GC I/O overheads, which suggests maintenance and GC’s overhead have different characteristics and have an overall similar impact.

Note that in maintenance of MeGA, if all chunks in a container are needed to be migrated to a new category, we can directly let this container belong to that new category without any chunk migration. It is interesting to observe that about 25.13% (WEB), 14.57% (CHM), 59.13% (SYN), and 72.95% (VMS) of containers do not need chunk migrations.

6 Conclusion

This paper proposes MeGA, a fine-grained deduplication framework, with three techniques: backup-workflow-oriented delta selector, delta-friendly data layout, and AFR traversing with delta prewriting, to address the three issues for different forms of poor locality caused by the introduction of delta

compression: reading base chunks, fragmentation, and repeatedly accessing containers, respectively. Evaluations show that MeGA achieves performance close to chunk-level deduplication while preserving fine-grained deduplication’s significant deduplication ratio advantage.

Acknowledgments

We are grateful to our shepherd and the anonymous reviewers for their insightful comments. This work was supported in part by NSFC under Grant 61972441, 61972112 and 61832004, in part by Shenzhen Science and Technology Program under Grants RCYX20210609104510007, JCYJ20210324131203009, JCYJ20200109113427092, and GXWD20201230155427003-20200821172511002, in part by Guangdong Basic and Applied Basic Research Foundation under Grant 2021A1515012634, 2021B1515020088 and in part by the HITSZ-J&A Joint Laboratory of Digital Design and Intelligent Fabrication under Grant no. HITSZ-J&A-2021A01.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

- [1] George Amvrosiadis and Medha Bhadkamkar. Identifying trends in enterprise data protection systems. In *Proceedings of the 2015 USENIX Annual Technical Conference*, 2015.
- [2] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2016.
- [3] Fabiano C. Botelho, Philip Shilane, Nitin Garg, and Windsor Hsu. Memory efficient sanitization of a deduplicated storage system. In *Proceedings of the 11th USENIX conference on File and Storage Technologies*, pages 81–94, 2013.
- [4] Andrei Z. Broder. On the resemblance and containment of documents. In *Proceedings of 1997 Compression and Complexity of SEQUENCES*, 1997.
- [5] Andrei Z. Broder. Identifying and filtering near-duplicate documents. In *Proceedings of the 11th Combinatorial Pattern Matching Annual Symposium*, 2000.
- [6] Zhichao Cao, Shiyong Liu, Fenggang Wu, Guohua Wang, Bingzhe Li, and David H. C. Du. Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, 2019.
- [7] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. CMD: classification-based memory deduplication through page access characteristics. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2014.
- [8] Liangfeng Cheng, Yuchong Hu, Zhaokang Ke, and Zhongjie Wu. Coupling right-provisioned cold storage data centers with deduplication. In *Proceedings of the the 50th International Conference on Parallel Processing*, 2021.
- [9] Fred Douglass, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano C. Botelho. The logic of physical garbage collection in deduplicating storage. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, pages 29–44, 2017.
- [10] Idilio Drago, Marco Mellia, Maurizio M. Munafò, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside drop-box: understanding personal cloud storage services. In *Proceedings of the 12th ACM SIGCOMM Internet Measurement Conference*, 2012.
- [11] Abhinav Duggal, Fani Jenkins, Philip Shilane, Ramprasad Chintheekindi, Ritesh Shah, and Mahesh Kamat. Data domain cloud tier: Backup here, backup there, deduplicated everywhere! In *Proceedings of the 2019 USENIX Annual Technical Conference*, 2019.
- [12] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Jingning Liu, Wen Xia, Fangting Huang, and Qing Liu. Reducing fragmentation for in-line deduplication backup storage via exploiting backup history and cache knowledge. *IEEE Trans. Parallel Distributed Syst.*, 27(3):855–868, 2016.
- [13] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *Proceedings of the 2014 USENIX Annual Technical Conference*, 2014.
- [14] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In *Proceedings of 2011 USENIX Annual Technical Conference*, pages 1–14, 2011.
- [15] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.
- [16] Navendu Jain, Michael Dahlin, and Renu Tewari. TAPER: tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the 3rd Conference on File and Storage Technologies*, 2005.
- [17] Keren Jin and Ethan L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of of SYSTOR 2009: The Israeli Experimental Systems Conference*, 2009.
- [18] Jonghwa Kim, Choonghyun Lee, Sang Yup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu-ung Lee, Sooyong Kang, Youjip Won, and Jaehyuk Cha. Deduplication in ssds: Model and quantitative analysis. In *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies*, 2012.
- [19] Keonwoo Kim, Jee-hong Kim, Changwoo Min, and Young Ik Eom. Content-based chunk placement scheme for decentralized deduplication on distributed file systems. In *Proceedings of the 13th International Conference on Computational Science and Its Applications*, 2013.

- [20] Ricardo Koller and Raju Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, 2010.
- [21] Lucas Kuhring and Zsolt István. Storing parquet tile by tile: Application-aware storage with deduplication. In *Proceedings of the 29th International Conference on Field Programmable Logic and Applications*, 2019.
- [22] Purushottam Kulkarni, Fred Douglass, Jason D. LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the 2004 USENIX Annual Technical Conference*, 2004.
- [23] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proceedings of the 11th USENIX conference on File and Storage Technologies*, 2013.
- [24] Xing Lin, Guanlin Lu, Fred Douglass, Philip Shilane, and Grant Wallace. Migratory compression: coarse-grained data reordering to improve compressibility. In *Proceedings of the 12th USENIX conference on File and Storage Technologies*, 2014.
- [25] Jian Liu, Yunpeng Chai, Chang Yan, and Xin Wang. A delayed container organization approach to improve restore speed for deduplication systems. *IEEE Trans. Parallel Distributed Syst.*, 27(9):2477–2491, 2016.
- [26] Joshua P. MacDonald. File system support for delta compression, 2000.
- [27] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, 2016.
- [28] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. *ACM Trans. Storage*, 7(4):14:1–14:20, 2012.
- [29] Jaehong Min, Daeyoung Yoon, and Youjip Won. Efficient deduplication techniques for modern backup operation. *IEEE Trans. Computers*, 60(6):824–840, 2011.
- [30] Jeffrey C. Mogul, Fred Douglass, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for http. *SIGCOMM Comput. Commun. Rev.*, 27(4):181–194, 1997.
- [31] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, 2001.
- [32] Mohammad Nasirifar and Angela Demke Brown. Deduplicating future data transfer using data exchanged in the past to decrease mobile bandwidth usage. In *Proceedings of the 18th Annual International Conference on Mobile Systems, Applications, and Services*, 2020.
- [33] Lars Nielsen, Dorian Burihabwa, Valerio Schiavoni, Pascal Felber, and Daniel E. Lucani. Minervafs: A user-space file system for generalised deduplication: (practical experience report). In *Proceedings of the 40th International Symposium on Reliable Distributed Systems*, 2021.
- [34] Sungbo Park, Ingab Kang, Yaebin Moon, Jung Ho Ahn, and G. Edward Suh. BCD deduplication: effective memory compression using partial cache-line deduplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.
- [35] Himabindu Pucha, David G. Andersen, and Michael Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, 2007.
- [36] Philip Shilane, Ravi Chitloor, and Uday Kiran Jonnalala. 99 deduplication problems. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems*, 2016.
- [37] Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. Wan-optimized replication of backup datasets using stream-informed delta compression. *ACM Trans. Storage*, 8(4):13:1–13:26, 2012.
- [38] Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. Delta compressed and deduplicated storage using stream-informed locality. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems*, 2012.
- [39] Mark W. Storer, Kevin M. Greenan, Darrell D. E. Long, and Ethan L. Miller. Secure data deduplication. In *Proceedings of the 2008 ACM Workshop On Storage Security And Survivability, StorageSS*, 2008.
- [40] Zhen Jason Sun, Geoff Kuenning, Sonam Mandal, Philip Shilane, Vasily Tarasov, Nong Xiao, and Erez Zadok. Cluster and single-node analysis of long-term deduplication patterns. *ACM Trans. Storage*, 14(2):13:1–13:27, 2018.
- [41] Haoliang Tan, Zhiyuan Zhang, Xiangyu Zou, Qing Liao, and Wen Xia. Exploring the potential of fast delta encoding: Marching to a higher compression ratio. In *Proceedings of the 2020 IEEE International Conference on Cluster Computing*, 2020.

- [42] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. Generating realistic datasets for deduplication analysis. In *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.
- [43] Grant Wallace, Fred Douglass, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, 2012.
- [44] Chunzhi Wang, Yanlin Fu, Junyi Yan, Xinyun Wu, Yucheng Zhang, Huiling Xia, and Ye Yuan. A cost-efficient resemblance detection scheme for post-deduplication delta compression in backup systems. *Concurrency and Computation: Practice and Experience*, 2021.
- [45] Avani Wildani, Ethan L. Miller, and Ohad Rodeh. HANDS: A heuristically arranged non-backup in-line deduplication system. In *Proceedings of the 29th IEEE International Conference on Data Engineering*, 2013.
- [46] Wen Xia, Hong Jiang, Dan Feng, Fred Douglass, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proc. IEEE*, 104(9):1681–1710, 2016.
- [47] Wen Xia, Hong Jiang, Dan Feng, and Lei Tian. DARE: A deduplication-aware resemblance detection and elimination scheme for data reduction with low overheads. *IEEE Trans. Computers*, 65(6):1692–1705, 2016.
- [48] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. Ddelta: A deduplication-inspired fast delta compression approach. *Perform. Evaluation*, 79:258–272, 2014.
- [49] Wen Xia, Chunguang Li, Hong Jiang, Dan Feng, Yu Hua, Leihua Qin, and Yucheng Zhang. Edelta: A word-enlarging based fast delta compression approach. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Storage and File Systems*, 2015.
- [50] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, and Gregory R. Ganger. Online deduplication for databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.
- [51] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, Jin Li, and Gregory R. Ganger. Reducing replication bandwidth for distributed document databases. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.
- [52] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, 2019.
- [53] Yucheng Zhang, Ye Yuan, Dan Feng, Chunzhi Wang, Xinyun Wu, Lingyu Yan, Deng Pan, and Shuanghong Wang. Improving restore performance for in-line backup system combining deduplication and delta compression. *IEEE Trans. Parallel Distributed Syst.*, 31(10):2302–2314, 2020.
- [54] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali Raza Butt. Duphunter: Flexible high-performance deduplication for docker registries. In *Proceedings of the 2020 USENIX Annual Technical Conference*, 2020.
- [55] Feng Zhou, Li Zhuang, Ben Y. Zhao, Ling Huang, Anthony D. Joseph, and John Kubiatowicz. Approximate object location and spam filtering on peer-to-peer systems. In *Proceedings of the 2003 ACM/IFIP/USENIX International Middleware Conference*, 2003.
- [56] Benjamin Zhu, Kai Li, and R. Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008.
- [57] Xiangyu Zou, Cai Deng, Wen Xia, Philip Shilane, Hao-liang Tan, Haijun Zhang, and Xuan Wang. Odess: Speeding up resemblance detection for redundancy elimination by fast content-defined sampling. In *Proceedings of the 37th IEEE International Conference on Data Engineering*, 2021.
- [58] Xiangyu Zou, Jingsong Yuan, Philip Shilane, Wen Xia, Haijun Zhang, and Xuan Wang. The dilemma between deduplication and locality: Can both be achieved? In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*, pages 171–185, 2021.

A Artifact Appendix

Abstract

The artifact is source code of a prototype deduplication system for backups that follows the ideas in the paper.

Scope

It could suggest the details and effectiveness of the delta selector, the delta-friendly data layout, the "Always-Forward-Reference" traversing, and the delta prewriting mechanism.

Contents

The artifact is source code of a prototype deduplication system for backups that follows the ideas in the paper. It mainly supports two main operations: (1) deduplicating and storing backup workloads and (2) restoring stored backup workloads.

Detailed manuals are introduced in our GitHub repository. In brief, the artifact supports the two operations with the following two commands.

```
=====  
# deduplicating and storing a new backup  
./MeGA --ConfigFile=[config file path] --task=write --InputFile=[backup work-  
load] --DeltaSelectorThreshold=[Delta Selector Threshold]  
  
# restoring a stored backup  
./MeGA --ConfigFile=[config file path] --task=restore --RestorePath=[path  
to restore] --RestoreRecipe=[which backup to restore (1 ~ n)]  
=====
```

MeGA generates several outputs when executing. Note that:

① MeGA includes chunk-level deduplication, delta compression, and local compression. The "total reduction ratio" suggests the benefits from all these parts on a single backup.

② The "total reduction ratio" simply indicates how many times the size of a single backup has been reduced. For the entire dataset, the user needs to add up the original size of all backups in a dataset and divide it by the "After Compression" of all backups to get the general "Dedup ratio" of the dataset, which is suggested in Figure 15.

③ The backup speed is related to the results in Figure 11.

④ The cache misses and average time cost are related to the results in Figure 12.

⑥ The arrangement duration is related to the results in Figure 17.

⑦ The restore speed is related to the results in Figure 13.

⑤ Figure 16 is just a general view, and it does not have new results.

Hosting

The source code is available at <https://github.com/Borelset/MeGA> (the "ContainerBased" branch).

Requirements

The Artifact has the following requirements.

Hardware Requirement:

- CPUs supporting AVX2 instructions.
- 32GB or larger RAM
- 7200rpm HDD drives for experiments.
- Another storage device for datasets. (400GB for full evaluations or 100GB for partly evaluations)

Software Requirement:

- `isal_crypto` [https://github.com/intel/isa-l_crypto]
- `jemalloc` [<https://github.com/jemalloc/jemalloc>]
- `openssl` [<https://github.com/openssl/openssl>]
- `zstd` [<https://github.com/facebook/zstd>]
- Reformat your HDD and deploy an XFS file system, as fragmentation of the file system will affect performance.



Secure and Lightweight Deduplicated Storage via Shielded Deduplication-Before-Encryption

Zuoru Yang[†], Jingwei Li^{‡,*}, and Patrick P. C. Lee[†]

[†]*The Chinese University of Hong Kong* [‡]*University of Electronic Science and Technology of China*

Abstract

Outsourced storage should fulfill confidentiality and storage efficiency for large-scale data management. Conventional approaches often combine encryption and deduplication based on deduplication-after-encryption (DaE), which first performs encryption followed by deduplication on encrypted data. We argue that DaE has fundamental limitations that lead to various drawbacks in performance, storage savings, and security in secure deduplication systems. In this paper, we study an unexplored paradigm called deduplication-before-encryption (DbE), which first performs deduplication and encrypts only non-duplicate data. DbE has the benefits of mitigating the performance and storage penalties caused by the management of duplicate data, but its deduplication process is no longer protected by encryption. To this end, we design DEBE, a shielded DbE-based deduplicated storage system that protects deduplication via Intel SGX. DEBE builds on frequency-based deduplication that first removes duplicates of frequent data in a space-constrained SGX enclave and then removes all remaining duplicates outside the enclave. Experiments show that DEBE outperforms state-of-the-art DaE approaches.

1 Introduction

Data outsourcing to public cloud storage provides a plausible solution for low-cost, large-scale data storage management in the face of explosive data growths [71]. To defend against data privacy leakage [57], clients require end-to-end encryption, such that their outsourced data be encrypted before being stored in (untrusted) public cloud storage. However, traditional symmetric encryption prohibits cross-user deduplication (i.e., removing duplicate data from multiple clients), since each client encrypts its own outsourced data with a distinct secret key, implying that the encrypted outputs from multiple clients are also distinct.

The literature has numerous studies (e.g., [3, 7, 8, 18, 23, 72, 74, 79]) on how to seamlessly combine encryption and deduplication for secure deduplicated storage in data outsourcing, which we collectively refer to as *deduplication-after-encryption (DaE)*. DaE first performs encryption on the outsourced data on the client side for confidentiality, followed by applying cross-user deduplication in the cloud to remove duplicate encrypted data for storage savings. To preserve the identical content after encryption, DaE encrypts data using a

symmetric key derived from the content of each *chunk* (the basic unit of deduplication), such that duplicate original chunks (called *plaintext chunks*) are always encrypted by the same key into duplicate encrypted chunks (called *ciphertext chunks*) that are later removed by deduplication.

Despite its popularity, we argue that DaE has fundamental limitations including high key management overhead, incompatibility with compression, and security risks (see §2.1 for details). Since DaE always manages a key for each chunk for encryption before deduplication, it not only unnecessarily generates a huge number of keys for duplicate chunks that will later be removed by deduplication, but also incurs high storage overhead for managing a huge number of keys for all duplicate and non-duplicate chunks [47]. In addition, DaE stores non-duplicate encrypted chunks, whose contents look randomized and have limited room for further space reduction from compression. Furthermore, DaE necessitates deterministic encryption to preserve the deduplication capability on ciphertext chunks. Such a deterministic nature is vulnerable to information leakage through frequency analysis [48, 49].

The limitations of DaE motivate us to explore a simple but unexplored paradigm called *deduplication-before-encryption (DbE)*, which first performs deduplication on the plaintext chunks and then encrypts the remaining non-duplicate plaintext chunks with any key that is independent of the chunk content. A major distinction from DaE is that DbE does not need to manage per-chunk keys for encryption/decryption, and we argue that DbE addresses the limitations of DaE (§2.2). However, DbE remains unexplored in secure deduplicated storage, mainly because the chunks are no longer protected by encryption in deduplication processing, which is carried out in the cloud for cross-user deduplication.

Our insight is that the deduplication process in DbE can be protected with *shielded execution* [4, 37]. To this end, we present DEBE, a shielded DbE-based deduplicated storage system with performance, storage savings, and security in mind. DEBE builds on Intel Software Guard Extensions (SGX) [41], which provides a shielded execution environment, called an *enclave*, for secure deduplication processing. A key challenge of realizing DEBE in SGX is the limited enclave space (e.g., up to 128 MiB [36]). Thus, we propose *frequency-based deduplication*, a two-phase deduplication scheme that can realize secure and lightweight deduplication with the space-constrained enclave. Specifically, DEBE first performs deduplication on the most frequent chunks inside an enclave,

*Corresponding author: Jingwei Li (jwli@uestc.edu.cn)

motivated by our observation that the most frequent chunks often contribute to a large fraction of duplicates in real-world backup workloads (§4.1). It then performs deduplication on the remaining less frequent chunks outside the enclave. With frequency-based deduplication, DEBE has the key advantages of: (i) high performance, as it removes most duplicates in the first-phase deduplication and incurs limited performance overhead for the second-phase deduplication outside the enclave; (ii) high storage savings via both deduplication and compression; and (iii) security, as it protects the most frequent chunks (which are more vulnerable to frequency analysis attacks [48]) inside the enclave.

We evaluate our DEBE prototype in a LAN testbed. DEBE achieves significant speedups over state-of-the-art DaE approaches (e.g., 10.09× and 13.08× speedups over DupLESS [7] in uploading non-duplicate and duplicate data, respectively). In our technical report [81], we also show that DEBE achieves high storage savings (e.g., 93.8% of key metadata storage savings compared with DaE) and reduces information leakage without compromising storage savings (e.g., by 87.7% of the relative entropy over TED [49], while TED incurs a storage blowup). The source code of our DEBE prototype is at: <https://github.com/yzr95924/DEBE>.

2 Background and Motivation

2.1 Limitations of Deduplication-after-Encryption

Deduplication is a widely deployed data reduction technique in modern storage [26, 27, 59, 77, 85]. We focus on *chunk-based deduplication*, which removes duplicates at the granularity of a *chunk*. Specifically, a deduplicated storage system partitions input file data into chunks. It identifies each chunk by a cryptographic hash (e.g., SHA-256), called a *fingerprint*, of the chunk content (assuming that fingerprint collisions of distinct chunks are practically impossible [10]). It maintains a key-value store, called the *fingerprint index*, to track the fingerprints of all existing stored chunks, and stores only the non-duplicate chunks. It also stores a manifest file, called the *file recipe*, for each file to track all chunks of the file in storage for file reconstruction. In addition, it may further apply *compression* to remove byte-level duplicates within the non-duplicate chunks for more storage savings [27, 73, 85].

Deduplication-after-encryption (DaE) combines deduplication and encryption for both confidentiality and storage savings. In DaE, a client locally encrypts the plaintext chunks and uploads the ciphertext chunks to the cloud, which then performs deduplication on the ciphertext chunks. One popular cryptographic primitive for DaE is *message-locked encryption (MLE)* [8], which formalizes that the key for chunk encryption/decryption is derived from the content of each chunk, so that identical plaintext chunks are always encrypted into identical ciphertext chunks for deduplication. An instantiation of MLE is *convergent encryption (CE)* [3, 18, 23, 72, 74, 79], which derives each chunk’s key based on its fingerprint.

CE is vulnerable to *offline brute-force attacks* [7], in which an adversary enumerates all possible plaintext chunks to derive their secret keys, attempts to decrypt a ciphertext chunk using each key, and deduces the plaintext chunk if the decryption succeeds. DupLESS [7] defends against offline brute-force attacks in CE via server-aided key management, by deploying a *key server* that generates the key of each chunk based on a global secret (securely owned by the key server) and the chunk fingerprint. Also, DupLESS implements key generation based on an oblivious pseudorandom function (OPRF) [63] to prevent the key server from learning the chunks or the keys during key generation, and rate-limits the key generation requests from clients to defend against *online brute-force attacks*, in which a malicious client aggressively issues key generation requests for different plaintext chunks to the key server.

Limitations. DaE is the state-of-the-art paradigm for building secure deduplicated storage systems. However, we argue that DaE suffers from three fundamental limitations.

- *L1 (High key management overhead).* DaE generates one key per chunk, leading to huge overheads for maintaining all chunk-based keys. Also, each client needs to encrypt its chunk-based keys via its own master secret key for protection. Thus, the key storage overhead increases proportionally with the numbers of chunks and clients, and is particularly significant for the workloads with high content redundancy (e.g., backups [77]) as they store only small amounts of non-duplicate data after deduplication. Also, DupLESS [7], which realizes server-aided key management, generates a key for the encryption of each chunk before the chunk is uploaded to the cloud, even though the chunk is a duplicate and is later removed by deduplication. As DupLESS employs OPRF and rate-limiting in key generation (see above), its key generation is shown to be expensive [70]. In short, DaE incurs high key management overhead, both in terms of key storage and key generation.
- *L2 (Incompatibility with compression).* In DaE, the cloud cannot further save additional storage space of non-duplicate encrypted chunks via compression, as encrypted chunks have high-entropy (almost random) contents. While a client may apply compression to the plaintext chunks before encryption and upload the encrypted compressed chunks, this leaks the compressed chunk lengths and introduces security risks [13].
- *L3 (Security risks).* Server-aided key management in DupLESS [7] makes the key server a single point-of-attack. If an adversary compromises the key server and has access to the global secret, it can infer the secret keys of chunks via offline brute-force attacks as in CE. Also, DaE is deterministic by nature and realizes one-to-one mappings between plaintext chunks and ciphertext chunks. An adversary can launch frequency analysis to infer the original plaintext chunks from the frequency distribution of ciphertext chunks in deduplicated storage [48].

2.2 Moving to Deduplication-before-Encryption

Given the limitations of DaE (§2.1), we study an unexplored paradigm, namely *deduplication-before-encryption (DbE)*, for secure deduplicated storage. Its idea is to first perform deduplication on the plaintext chunks to remove duplicates, followed by encrypting the non-duplicate plaintext chunks into ciphertext chunks for storage.

DbE naturally offers several benefits over DaE. First, since deduplication is applied first, DbE can encrypt each non-duplicate plaintext chunk with a content-independent key as in traditional symmetric encryption (§1) without compromising deduplication. This avoids generating and storing per-chunk content-derived keys and reduces the key management overhead (i.e., L1 addressed). Second, DbE can apply compression to the non-duplicate plaintext chunks after deduplication for further storage savings, followed by encrypting the compressed non-duplicate plaintext chunks (i.e., L2 addressed). Finally, since DbE can perform encryption with a content-independent key, it no longer needs a key server for per-chunk key generation as in DupLESS. This removes the single point-of-attack in the key server (i.e., L3 addressed).

The major challenge of DbE, however, is to decide whether clients or the cloud should perform deduplication, which is no longer protected by encryption. We consider three scenarios:

- Each client maintains a local fingerprint index for its own plaintext chunks. It encrypts the non-duplicate plaintext chunks and uploads the ciphertext chunks to the cloud. However, this approach prohibits cross-user deduplication.
- The cloud maintains a global fingerprint index to track the stored chunks of all clients. Each client first submits the fingerprints of its own plaintext chunks to the cloud to query if they can be deduplicated. It encrypts the non-duplicate plaintext chunks identified by the cloud, and uploads the ciphertext chunks to the cloud. This approach, also referred to as *source-based deduplication* [35], is vulnerable to *side-channel attacks* [35,62] since any malicious client can infer if some target chunk has already been stored by querying if the target chunk can be deduplicated.
- Each client uploads all chunks to the cloud. The cloud performs deduplication based on its global fingerprint index that tracks the stored chunks of all clients, followed by encrypting the non-duplicate chunks. This approach, also referred to as *target-based deduplication* [35], hides the deduplication pattern from the clients and is secure against side-channel attacks. However, each client inevitably exposes its plaintext chunks to the cloud.

Thus, DbE remains unexplored in the literature, while existing studies mostly focus on DaE for secure deduplicated storage.

2.3 Intel SGX

In this work, we realize DbE with target-based deduplication and show how we protect DbE via *shielded execution*. We implement shielded execution using Intel SGX [41]. As our

major requirement is to provide a secure memory region for data processing in the untrusted cloud, we conjecture that our design can be supported with other shielded execution technologies (e.g., ARM TrustZone [67] and AMD SEV [2]).

SGX basics. SGX is a set of extended instructions for Intel CPUs to realize a shielded execution environment, called an *enclave*, in an encrypted and integrity-protected memory region called the *enclave page cache (EPC)*. It ensures confidentiality and integrity for in-enclave contents with hardware protection. It provides two interfaces to interact with untrusted applications outside the enclave: (i) *enclave calls (ECalls)*, which permit applications to safely access in-enclave contents, and (ii) *outside calls (OCalls)*, which allow in-enclave code to issue function calls in applications.

Challenges. Realizing DbE in SGX is non-trivial due to the resource constraints of an enclave. First, the EPC size is limited (e.g., up to 128 MiB [36]). When an enclave has memory usage exceeding the EPC size, it encrypts and evicts the unused memory pages to the unprotected main memory, and decrypts and verifies the integrity of the evicted pages when loading them back to the EPC. This incurs expensive EPC paging overhead [5, 21]. Although recent SGX designs support a large EPC size of up to 1 TiB [44], they provide weaker security guarantees due to the loss of integrity tree protection [28]. Second, both ECalls and OCalls involve expensive hardware operations (e.g., flushing TLB entries [5]) that lead to significant context switching overhead (e.g., around 8,000 CPU cycles per call [66, 78]).

3 Design Overview

3.1 DEBE Architecture

We make a case for DbE by designing DEBE, a shielded DbE-based deduplicated storage system based on Intel SGX [41]. Figure 1 presents the architecture of DEBE; note that DEBE does not maintain a key server as in DupLESS [7] (§2.1). We consider a *multi-tenant* scenario, in which the clients from different organizations store outsourced data to a cloud storage service (or the cloud in short). DEBE performs target-based deduplication [35] (§2.2) to remove the duplicate data of multiple clients in the cloud. Currently, each DEBE client uploads all its data to the cloud for deduplication. Although a client may apply deduplication to its own data to save upload bandwidth without introducing side-channel attacks [50], our design does not make this assumption.

To prevent the cloud from accessing any plaintext chunks during deduplication processing, DEBE hosts an enclave in the cloud and performs deduplication inside the enclave. To support the multi-tenant scenario, we assume that a trusted third party (e.g., a certificate authority in the public key infrastructure (PKI) [56]) is responsible for the enclave setup. Specifically, the trusted third party compiles the enclave code into a shared object (as a .so file). It distributes the shared object to the cloud, along with its signature for integrity ver-

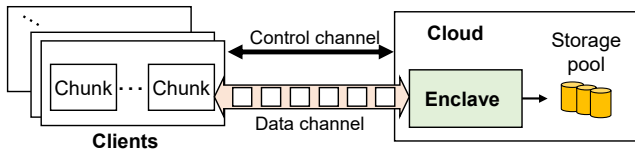


Figure 1: DEBE architecture.

ification. The cloud loads the shared object to bootstrap the enclave. The trusted third party can initiate *remote attestation* [41] to ensure that the correct code is loaded into the enclave, and it can go offline after the enclave is bootstrapped.

After the enclave is bootstrapped, each client sets up two secure communication channels: (i) the *control channel* with the cloud for transmitting the commands of storage operations and (ii) the *data channel* with the enclave for transmitting the plaintext chunks originated by the client. Currently, DEBE sets up the control channel between a client and the cloud using traditional SSL/TLS authentication. To set up the data channel between a client and the enclave, since the enclave cannot directly access the network socket of the cloud [41], DEBE implements the Diffie-Hellman key exchange to agree on a session key between a client and the enclave (§4.2), and the session key is used to protect the data channel. Note that other key exchange algorithms can be used for session key establishment.

To upload a file to the cloud, a client divides the file data into fixed-size or variable-size plaintext chunks as in traditional chunk-based deduplication (§2.1). It issues an upload request to the cloud through the control channel, and sends all plaintext chunks to the enclave through the data channel. The enclave deduplicates and compresses the received plaintext chunks on a per-batch basis (§4.1), encrypts the remaining non-duplicate compressed chunks into ciphertext chunks, and emits the ciphertext chunks and the file recipe to the storage pool.

To download a file, the client issues a download request to the cloud through the control channel. The enclave then retrieves the file’s recipe and the corresponding ciphertext chunks. Finally, it decrypts the ciphertext chunks, and decompresses and returns the plaintext chunks to the client through the data channel.

Practical relevance of DEBE. DEBE focuses on multi-tenant deduplication, which is widely deployed in practice (e.g., Dropbox [24], Druva [25], Cohesity [14], and Memopal [58]) and is shown to achieve higher storage savings than single-tenant deduplication by removing the duplicate data from multiple clients [50,59,83]. Existing DaE approaches are also designed for multi-tenant deduplication, while DEBE addresses the limitations of DaE (§2.1). Although DEBE incurs costs due to shielded execution (e.g., the enclave verification fee from a trusted third party), its improvements over DaE approaches (in terms of performance, storage savings, and robustness; see §3.3) provide incentives for a cloud storage provider to use DEBE to provide secure and cost-effective cloud storage services for customers.

3.2 Threat Model

We consider an honest-but-curious adversary that does not modify the system protocol but aims to compromise data confidentiality by identifying the original content of the out-sourced data stored in the cloud. The adversary can tap into the cloud and gain access to any data stored in the unprotected main memory of the cloud as well as the ciphertext chunks in the storage pool. It can also eavesdrop on the content of OCalls issued to the unprotected main memory (e.g., the parameters and untrusted functions used by OCalls).

Our threat model assumes that the enclave is trusted and reliable; its authenticity is verified by remote attestation [41] when it is created (§3.1). Any denial-of-service or side-channel attack against SGX is protected by existing solutions [64, 76]. Also, if the adversary has access to a compromised client, then it can access all the plaintext chunks of the client. However, since DEBE performs target-based deduplication (§3.1), the adversary cannot access or infer the plaintext chunks of other non-compromised clients.

3.3 Design Goals

DEBE is designed for clients from multiple tenants (§3.1) to securely outsource their storage management to public cloud storage services. It targets storage workloads with high content redundancy (e.g., backups [77] and file system snapshots [59]) that can be effectively removed by deduplication and compression. DEBE has the following design goals:

- *High performance.* DEBE has significantly lower key management overhead than DaE approaches. It also incurs limited overhead in SGX.
- *High storage savings.* DEBE supports *exact* deduplication (§4.4), i.e., all duplicates from multiple clients can be removed. It also applies compression to the non-duplicate chunks after deduplication for extra storage savings.
- *Confidentiality.* DEBE preserves the security of DaE by enforcing end-to-end encryption for the plaintext chunks between each client and the enclave, and the plaintext chunks are inaccessible by the cloud provided that the enclave is trusted and reliable (§3.2). DEBE remains secure against offline brute-force attacks in CE (§2.1), without the need of server-aided key management as in DupLESS [7].
- *Robustness over DaE.* DEBE mitigates the single point-of-attack of DaE by eliminating the key server. It also mitigates the information leakage caused by frequency analysis against DaE [48, 49].

4 Detailed Design

4.1 Main Idea

DEBE’s core idea is to perform deduplication inside the enclave (hosted in the cloud), so as to provide confidentiality guarantees for the plaintext chunks during the deduplication process. Keeping a full fingerprint index (or the *full index* in short) inside the enclave can track the fingerprints of all

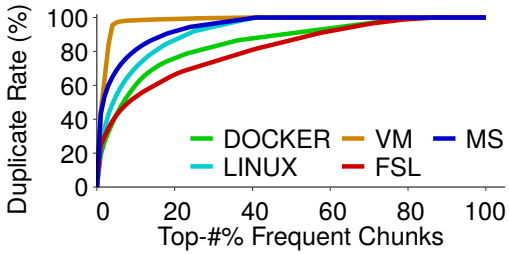


Figure 2: Duplicate rate versus top-percentage of frequent chunks in five real-world traces.

non-duplicate chunks being stored, but incurs significant EPC paging overhead due to the limited EPC size (§2.3). Alternatively, managing the full index outside the enclave saves the EPC usage, but incurs expensive context switching due to excessive OCalls for querying the full index (§2.3).

We propose *frequency-based deduplication*, which performs secure deduplication subject to the resource constraints of the enclave. Our insight is that the frequencies (i.e., numbers of duplicates) of chunks are highly skewed in practical backup workloads, such that *a small fraction of chunks can contribute to a large fraction of duplicates*. To justify, we conduct trace analysis on five real-world backup traces (see §6.1 for the trace details). We measure the *duplicate rate* for a subset of input chunks, defined as the ratio between the total size of duplicate chunks derived from the subset of chunks and the total size of duplicate chunks in the whole trace (note that a chunk is said to be a duplicate chunk if its identical copy has already been stored and it can be removed by deduplication). Figure 2 shows the duplicate rate versus the top-percentage of frequent chunks (ranked by their frequencies in descending order). For example, in the VM trace, the top-5% of frequent chunks contribute to a duplicate rate of around 97%. This implies that if we maintain a small fingerprint index to track the top-5% of frequent chunks, we can remove around 97% of duplicate data and achieve high storage savings.

The idea of frequency-based deduplication is to separate the deduplication process based on chunk frequencies. It manages a small fingerprint index inside the enclave to remove the duplicates from the most frequent chunks. It also maintains the full index outside the enclave to remove the remaining duplicates for the less frequent chunks. Frequency-based deduplication addresses both performance and security concerns. For performance, it only manages a small fingerprint index for the most frequent chunks inside the enclave to remove a large fraction of duplicate chunks. Thus, it mitigates the EPC paging overhead. It also reduces the context switching overhead as it only queries the full index outside the enclave via OCalls for a limited fraction of less frequent chunks. For security, since the most frequent chunks are more vulnerable to frequency analysis [48], we remove the duplicates of the most frequent chunks with in-enclave processing only. Thus, an adversary in the cloud cannot readily learn the frequencies of the most frequent chunks, and hence the information

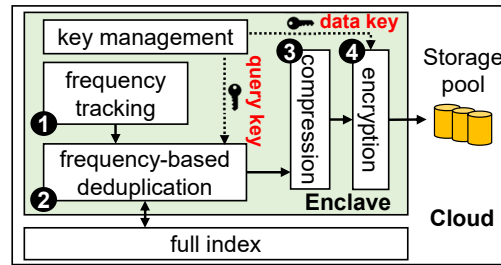


Figure 3: Architecture of the enclave.

leakage caused by frequency analysis is limited.

Enclave architecture and design roadmap. Figure 3 depicts the architecture of the enclave in DEBE. Initially, the enclave is bootstrapped with a set of keys and establishes secure data channels with each client (§4.2). Then the enclave tracks the frequency of each plaintext chunk received from the data channel of a client (§4.3). Based on the chunk frequencies, frequency-based deduplication removes the duplicates of the most frequent plaintext chunks and interacts with the full index outside the enclave to remove the duplicates of the remaining less frequent plaintext chunks (§4.4). The enclave performs compression on the non-duplicate plaintext chunks and encrypts the compressed plaintext chunks. Finally, the enclave stores the ciphertext chunks in the storage pool (§4.5).

4.2 Key Management

The enclave maintains a set of keys for the secure storage of chunks after deduplication and compression as well as for secure communication with clients.

Data key and query key. The enclave maintains two long-term keys, which remain valid throughout the lifetime of the enclave (i.e., the whole duration when DEBE is running): (i) the *data key* for encrypting and decrypting the compressed non-duplicate plaintext chunks in secure storage, and (ii) the *query key* for protecting the information of plaintext chunks when querying the full index outside the enclave (§4.4). When the enclave is bootstrapped, it initializes both the data key and the query key via the on-chip hardware random number generator (i.e., `sgx_read_rand` [42]). Both keys can be periodically renewed via existing approaches (e.g., key regression [30]), without compromising deduplication as DEBE performs deduplication before encryption.

Session key. Recall that each client maintains a data channel with the enclave for secure data communication (while maintaining a control channel with the cloud for securely issuing storage operations) (§3.1). Each data channel protects its communication using a short-term *session key*, which remains valid for a single communication session. It establishes a session key for the data channel using Diffie-Hellman key exchange through the control channel. The session key is kept in the enclave during the communication session of the client, and will be freed after the session is completed (both the control and data channels will be released as well).

Per-client master key. The enclave requires each client to

submit a *master key* through the data channel for each storage request. It uses the master key to protect the file recipes for the client’s files and enforces the client’s ownership of the files. Similar to the session keys, the enclave only keeps the master key of the client for a single communication session and will destroy the master key at the end of the session, so the storage overhead for the master keys is also limited.

4.3 Frequency Tracking

The enclave needs to track the frequencies of plaintext chunks to identify the most frequent and less frequent chunks for frequency-based deduplication. To mitigate the EPC usage (§2.3), the enclave uses a *Count-Min Sketch (CM-Sketch)* [16] to track the *approximate* frequency of each chunk with fixed-size space and small errors.

The CM-Sketch is a two-dimensional array with r rows of w counters each. One key design here is to limit the computational overhead of mapping the plaintext chunks to the counters. To do so, our insight is that the chunk fingerprint is computed as a cryptographic hash (e.g., SHA-256 in our case), so we can treat the chunk fingerprint as a random input value and map it directly to a counter without compromising the accuracy of the CM-Sketch. Specifically, for each plaintext chunk M , the enclave partitions the fingerprint of M into r pieces. It takes the i -th piece modulo w to find one of the w counters, indexed from 0 to $w - 1$, in row i ($1 \leq i \leq r$) and increments each of the mapped counters by one; this is in contrast to the traditional CM-Sketch, which maps the input to the counters of different rows using pairwise independent hash functions [16] and hence has extra computational overhead. To estimate the frequency of a chunk, the enclave uses the minimum value of the r mapped counters of the chunk. By default, we configure $r = 4$, $w = 256$ K, and 4-byte counters, so the overall EPC usage of the CM-Sketch is only 4 MiB.

4.4 Frequency-based Deduplication

We present the design of frequency-based deduplication, which removes all duplicate plaintext chunks in two phases based on their estimated frequencies (§4.3).

First-phase deduplication. The enclave maintains a small fingerprint index, called the *top- k index*, to deduplicate the k most frequent plaintext chunks. We implement the top- k index as a combination of a min-heap and a hash table, as shown in Figure 4. The min-heap differentiates the top- k -frequent and less frequent plaintext chunks. It tracks top- k estimated frequencies of the plaintext chunks, such that the root heap entry corresponds to the plaintext chunk with the minimum frequency in the current top- k estimated frequencies. Each heap entry in the min-heap stores a pointer to a hash entry in the hash table. On the other hand, the hash table is used for duplicate detection, as in conventional deduplication. Each hash entry stores a mapping from the chunk fingerprint to a tuple of elements: (i) the pointer to the heap entry (i.e., both the heap entry and the hash entry reference each other), (ii)

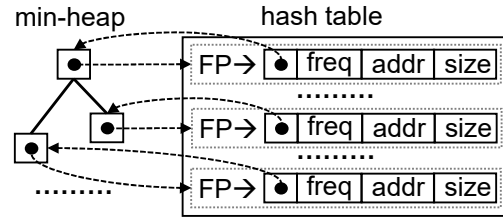


Figure 4: Overview of the top- k index.

the estimated frequency of the chunk, (iii) the chunk address (including the container ID and the internal offset within the container; see §4.5), and (iv) the compressed chunk size (i.e., the size of the chunk after compression).

Given a plaintext chunk, to perform the first-phase deduplication, the enclave takes the estimated frequency of the plaintext chunk obtained from the CM-Sketch (§4.3) and the chunk fingerprint as inputs. It first checks against the root heap entry of the min-heap. If the input frequency is smaller than the minimum frequency of the min-heap (i.e., the chunk is a less frequent chunk), the enclave skips querying the hash table for the chunk and proceeds to the second-phase deduplication (see below); otherwise (i.e., the chunk is a top- k -frequent chunk), the enclave uses the input fingerprint to look up the hash table. We have the following two cases:

- If the fingerprint is found in the hash table (i.e., the chunk is a duplicate), the enclave updates the frequency in the hash table and adds both the chunk address and the compressed chunk size to the file recipe (§4.5). Since the frequency is updated, it also adjusts the min-heap based on the pointer to the heap entry in the min-heap.
- If the fingerprint is not found in the hash table (i.e., the chunk is a new top- k -frequent chunk), the enclave creates a new hash entry in the hash table and inserts a new heap entry containing the pointer to the new hash entry into the min-heap. If the min-heap has already stored k heap entries, the enclave deletes the current root heap entry of the min-heap (with the minimum frequency) and also deletes the corresponding hash entry in the hash table via the pointer stored in the root heap entry. Since the chunk may have already been stored (but not tracked by the top- k index as its frequency is low), the enclave also runs the second-phase deduplication on the chunk and updates the chunk address and the compressed chunk size according to the result of the second-phase deduplication.

We show that the top- k index has low space usage. Suppose that the chunk fingerprint has 32 bytes (a SHA-256 hash), the chunk address has 12 bytes (an 8-byte container ID and a 4-byte internal offset; see §4.5), and the compressed chunk size has 4 bytes. For each top- k -frequent chunk, the hash entry additionally stores a 4-byte frequency and a pointer to a heap entry. Since we implement the min-heap as an array, the pointer to a heap entry can be represented as a 4-byte integer array index. Also, the heap entry keeps an 8-byte pointer to a hash entry. Overall, each top- k -frequent chunk uses 64 bytes

in the top- k index (excluding the internal pointers of the hash table, which we now implement as an `unordered_map` of the C++ standard library). For example, to track 512 K most frequent chunks, the EPC usage of the top- k index is 32 MiB.

We further show that the top- k index has low time complexity. For each plaintext chunk, the top- k index can return the minimum frequency (from the root heap entry) in the current min-heap in constant time. For a top- k -frequent chunk, the top- k index needs to further check the hash table (in constant time) and update the min-heap. Since we store the pointer to the heap entry in the hash entry, we can directly update the corresponding heap entry when the frequency is changed, without searching the whole min-heap for its location. Thus, the time complexity of updating the min-heap is $\mathcal{O}(\log k)$.

Second-phase deduplication. The second-phase deduplication is performed on the plaintext chunks that are not removed by the first-phase deduplication, including the less frequent chunks and the fresh new top- k -frequent chunks whose fingerprints are new to the top- k index. DEBE manages a full index outside the enclave as the EPC size is limited (§2.3). We implement the full index as a hash table, in which each entry stores the mapping from the *encrypted fingerprint* of a plaintext chunk to the *encrypted chunk information* (i.e., the chunk address and the compressed chunk size, both of which are encrypted by the query key) of the corresponding ciphertext chunk. Our rationale of encrypting both the fingerprint and the chunk information is to prevent any adversary in the cloud from inferring the plaintext chunks, since the full index is not protected by the enclave.

Given a plaintext chunk, to perform the second-phase deduplication, the enclave *deterministically* encrypts the fingerprint of the plaintext chunk (not removed by the first-phase deduplication) with the query key (§4.2), so that duplicate plaintext chunks from different clients are always mapped to duplicate encrypted fingerprints for cross-user deduplication. It then queries the full index based on the encrypted fingerprint via an OCall. If the encrypted fingerprint is found in the full index, the OCall returns the encrypted chunk information that will be decrypted by the query key inside the enclave. Then the enclave will update the address and the compressed chunk size into the file recipe (§4.5). Otherwise, if the encrypted fingerprint is new to the full index, the enclave identifies this chunk as a non-duplicate chunk, assigns the chunk an address, compresses the chunk to obtain its compressed chunk size, and encrypts both the address and the compressed chunk size with the query key. It then updates the encrypted fingerprint and the corresponding encrypted chunk information in the full index. Note that the context switching overhead due to OCalls is limited, as a large fraction of duplicates are expected to have been removed by the first-phase deduplication.

Remarks. Traditional efficient indexing techniques for deduplication, such as similarity-based [9] and locality-based deduplication [52] approaches, can also address the limited EPC size by loading only a portion of the full index into the en-

clave. However, they only achieve *near-exact* deduplication (i.e., some duplicates cannot be removed), while DEBE can achieve exact deduplication (§3.3).

Note that the CM-Sketch may overestimate the chunk frequencies as multiple chunks can be mapped to the same counters (§4.3). Thus, the enclave may track some less frequent chunks in the top- k index. Nevertheless, it does not affect the storage savings from deduplication, as the full index still tracks all currently stored non-duplicate chunks.

4.5 Storage Management

Container storage. DEBE manages physical chunks in fixed-size *containers* to mitigate disk I/O costs [51]. The enclave performs compression on the non-duplicate plaintext chunks after deduplication, and encrypts the compressed non-duplicate plaintext chunks into ciphertext chunks with the data key. It writes each ciphertext chunk, together with an initialization vector (IV) (§5), into an in-memory container inside the enclave. When the in-memory container is full, the enclave emits it to persistent storage in the cloud. Note that DEBE only stores an IV (of size 16 bytes in AES-256) for each non-duplicate ciphertext chunk *after* deduplication, while DaE approaches store an encrypted key (of size 32 bytes in AES-256) for each ciphertext chunk *before* deduplication and incurs substantial key storage overhead when there exist many duplicate chunks (§2.1).

Also, the enclave creates and manages the file recipe for each uploaded file. Each entry in the file recipe keeps the chunk address and the compressed chunk size of each ciphertext chunk of the file. Note that when the enclave adds entries to the file recipe, it does not need to perform compression for the duplicate chunk to obtain its compressed chunk size, since the compressed chunk size has been stored in both the top- k index and the full index. To guarantee the ownership of the file, the enclave encrypts the file recipe by the client's master key and stores the encrypted file recipe as a regular file. Since the enclave treats containers (each of which contains multiple ciphertext chunks) as the basic I/O units and the chunk size is stored in the file recipe (protected by the per-user master key), DEBE preserves the security of compression as it avoids leaking the lengths of compressed chunks [13].

Downloads. To download a file, the client issues a download request and its master key to the enclave through the secure data channel. The enclave retrieves the file recipe and decrypts the file recipe with the given master key. It then parses the decrypted file recipe to obtain the chunk addresses and compressed chunk sizes. To restore all chunks, the enclave exposes the container IDs of the requested chunks to the cloud to perform I/Os via OCalls. Once the cloud fetches the corresponding containers into the unprotected main memory, the enclave accesses the ciphertext chunks based on their internal offsets and decrypts the ciphertext chunks by the data key. Finally, it decompresses and sends the plaintext chunks to the client through the data channel.

4.6 Security Discussion

We discuss the security of DEBE in response to our threat model (§3.2). We focus on two cases.

Case 1: A snapshot adversary gains one-time access to contents in unprotected memory and storage pool. DEBE enforces end-to-end encryption for the plaintext chunks between each client and the enclave, and provides *semantic security* [33] for the ciphertext chunks stored in the cloud. Specifically, it sets up a secure data channel that encrypts all plaintext chunks exchanged between a client and the enclave by a session key. It performs deduplication inside the enclave (that is oblivious to the adversary), and encrypts the non-duplicate plaintext chunks into ciphertext chunks by the data key before the ciphertext chunks are stored. Note that both data-in-transit and data-at-rest encryption operations are based on traditional symmetric encryption, and semantic security is achieved.

Case 2: A persistent adversary eavesdrops on OCalls in deduplication. DEBE encrypts both chunk fingerprints and chunk information by the query key inside the enclave before it includes them as the inputs of OCalls for accessing the full index outside the enclave. Thus, even though an adversary can eavesdrop on the OCalls, it cannot infer the original inputs from the OCalls.

One potential information leakage is that a persistent adversary (that stays in the cloud for a long time) can learn the chunk frequencies in the deduplication process, as the enclave maps duplicate plaintext chunks into duplicate encrypted fingerprints when querying the full index. Specifically, the adversary can track the frequency distribution of encrypted fingerprints by monitoring the OCalls, and launch frequency analysis to infer the plaintext chunks. However, DEBE limits such information leakage to the less frequent chunks, which are relatively robust against frequency analysis [48] (for comparisons, DaE leaks the frequencies of *all* chunks since it is deterministic by nature; see §2.1). Our evaluation shows that DEBE mitigates information leakage more effectively than TED [49], a state-of-the-art approach that trades storage savings for security (see our technical report [81]).

Remarks. A powerful adversary may launch frequency-based side-channel attacks by simultaneously compromising a client and the cloud. If it proactively lets the compromised client upload artificial chunks to the cloud and monitors OCalls in the cloud, the adversary could infer chunk frequencies and even identify the most frequent chunks among the clients. While the practical damage caused by such side-channel attacks remains an open question, we can obfuscate the chunk frequency information by perturbing the OCalls patterns (e.g., adding dummy OCalls), at the expense of incurring extra performance overhead.

5 Implementation

We have implemented a prototype of DEBE in C++ on Linux based on Intel SGX SDK Linux 2.7 [42]. It uses OpenSSL-

1.1.1 [65] and Intel SGX SSL [43] for cryptographic operations. Our current prototype contains 17.5 K LoC.

Each client implements FastCDC [80] to realize content-defined chunking, where the minimum, average, and maximum chunk sizes are configured at 4 KiB, 8 KiB, and 16 KiB, respectively. The container size is 4 MiB. We implement Diffie-Hellman key exchange based on NIST P-256 elliptic curve for session key management of the data channel between the client and the enclave. The enclave computes the fingerprints of plaintext chunks via SHA-256 and encrypts each unique plaintext chunk via AES-256 in GCM mode with a random 16-byte IV. Also, it encrypts the fingerprint of each plaintext chunk via AES-256 in CMC mode with a 16-byte zero IV to support queries over encrypted fingerprints (as in CryptDB [68]). Both SHA-256 and AES-256 are configured to use (via OpenSSL EVP API) the Intel New Instructions Set for hardware-accelerated operations [39, 40]. We also implement LZ4 [15] for lossless stream-based compression in the enclave for chunk compression after deduplication.

To mitigate context switching overhead of the enclave, DEBE transmits and processes chunks on a per-batch basis (the default batch size is 128 chunks). Also, to speed up downloads, the cloud keeps an in-memory least-recently-used cache (256 MiB by default) to hold the recently accessed containers. For each container access request issued by the enclave (§4.5), the cloud first checks the cache and retrieves the containers from local storage only if they are not in cache. **Limitations.** DEBE currently does not address crash consistency. We now discuss how to extend DEBE with crash consistency, and pose the implementation as future work.

When a system crash occurs, DEBE would lose its in-enclave contents (i.e., the query key, the data key, the CM-Sketch, the top-*k* index, and the in-memory container pending to be persisted into the storage pool). We can extend DEBE to recover the query and data keys, the CM-Sketch, and the top-*k* index via *sealing*, an SGX feature that encrypts in-enclave content for secure out-enclave storage on disk [41]. When the enclave is bootstrapped (§3.1), DEBE stores a persistent copy of both the query key and data key by sealing. Also, it makes periodic snapshots of the CM-Sketch and top-*k* index by sealing. To restore the enclave states from a system crash, the cloud re-initializes the enclave by unsealing the keys and snapshots back to the enclave.

To realize crash consistency, we can augment DEBE with write-ahead logging [61] to record updates in on-disk logs before updating the in-memory CM-Sketch and top-*k* index. To recover from the data loss of the in-memory container, the enclave can log the IDs of the persisted containers for the currently uploaded file in on-disk logs. If a system crash occurs during the current upload, DEBE can roll back to the state before the upload starts based on the logs. It finally notifies the client to re-upload the file. Note that logging the changes into on-disk logs would incur extra OCalls. To mitigate the context switching overhead of logging, we can

batch multiple logging operations in a single OCall.

We can initialize a new CM-Sketch and a new top- k index after enclave recovery. This would not affect the storage savings from deduplication, provided that the full index is crash-consistent (e.g., via its implementation in a persistent key-value store) and tracks all currently stored non-duplicate chunks. However, DEBE incurs extra performance overhead, as it cannot learn frequent chunks and hence incurs more OCalls to build the top- k index from scratch.

6 Evaluation

We deploy DEBE in a local cluster of 11 machines connected with 10 GbE. Each machine has a quad-core 3.4 GHz Intel Core i5-7500 CPU and 32 GiB RAM, and is installed with Ubuntu 16.04. We deploy one or multiple clients, a key server (for DaE only), or a cloud storage backend on distinct machines. The machine for the cloud storage backend is attached with a TOSHIBA DT01ACA 1 TiB 7200 rpm SATA hard disk. By default, DEBE sets $k=512$ K for the top- k index and configures the CM-Sketch with $r=4$ and $w=256$ K, so as to keep the overall EPC usage within 64 MiB to limit the paging overhead in SGX. Note that we can tune the parameters based on the available EPC size.

We evaluate DEBE using both synthetic and real-world datasets. We summarize our evaluation results as follows.

- DEBE accelerates the uploads of non-duplicate and duplicate data of state-of-the-art DaE approaches by up to $10.09\times$ and $13.08\times$, respectively (Exp#1). Its frequency-based deduplication only takes 5.8-18.4% of the overall upload time (Exp#2). It preserves high performance for multi-client uploads/downloads (Exp#3) and various synthetic workloads (Exp#4).
- For real-world workloads, DEBE achieves 1.17 - $2.76\times$ speedups over state-of-the-art deduplication alternatives (Exp#5), and preserves high performance in long-term uploads and downloads (Exp#6).

In our technical report [81], we present additional evaluation results and show that DEBE achieves high storage savings and preserves security against frequency analysis.

6.1 Datasets

Synthetic datasets. We consider two synthetic datasets for our evaluation. The first dataset, namely *SYN-Unique*, includes non-duplicate and individually compressible chunks. Specifically, we generate SYN-Unique as a set of 2 GiB compressible files via the LZ data generator [38], which generates synthetic data based on SDGen [34]. The LZ data generator takes two parameters as inputs: (i) a compression ratio, which specifies the compressibility of the generated data, and (ii) a random seed for data generation. We configure the compression ratio as 2 to resemble real-world backup workloads [77], and vary the random seeds to generate distinct synthetic files. We perform chunking on each synthetic file

to ensure that its chunks are globally unique over all files. We use the dataset for stress-testing different schemes with non-duplicate chunks.

The second dataset, namely *SYN-Freq*, includes the original chunks (before deduplication) following a target frequency distribution. We build a synthetic file generator that generates files whose chunk frequencies follow a Zipf distribution as shown by prior work [83, 84]. Our generator takes three parameters as inputs: (i) the number of original chunks, (ii) the deduplication ratio (i.e., the ratio between the original data size and the non-duplicate data size), and (iii) the Zipfian constant (a larger constant implies higher skewness). To generate a synthetic file, we prepare a set of non-duplicate 48-bit fingerprints based on the expected number of non-duplicate chunks (i.e., the number of original chunks divided by the deduplication ratio). We assign each fingerprint with a compression ratio based on the normal distribution with a mean of 2 and a variance of 0.25 [46, 77]. To generate each original chunk, we sample its fingerprint from the fingerprint set based on the target Zipf distribution, and construct its content using the LZ data generator [38] with the compression ratio and fingerprint (as the random seed) as inputs. Finally, we generate the SYN-Freq dataset as a set of synthetic files, each of which contains 13,107,200 8-KiB original chunks (i.e., 100 GiB) and a deduplication ratio of $5\times$. The number of non-duplicate chunks is large enough that only a subset of non-duplicate chunks can be tracked by the top- k index.

Real-world datasets. We consider five real-world datasets of backup workloads, which are also used in previous studies for trace-driven evaluation [49, 50, 69, 70, 80, 86]:

- *DOCKER*: docker snapshots (from v4.1.0 to v7.0.0) of Couchbase [17] from Docker Hub [22];
- *LINUX*: snapshots (from stable versions between v2.6.13 and v5.9) of Linux source code [54], in which each snapshot is stored in the *mtar* format [53];
- *FSL*: home directory snapshots [29], among which we select 42 snapshots from nine users in 2013;
- *MS*: Windows file system snapshots [59], among which we select 30 snapshots of size around 100 GiB each; and
- *VM*: virtual machine snapshots [50] collected by ourself.

Table 1 shows the statistics of the five real-world datasets. Previous studies have shown that multi-tenant deduplication can achieve higher storage savings than single-tenant deduplication in FSL, MS, and VM [50, 59, 75]. Given the limited available disk space in our testbed, we sample a subset of snapshots from the original datasets [29, 59] for FSL and MS as in [49]. As FSL, MS, and VM only contain fingerprints, we generate compressible chunk contents as in SYN-Freq.

6.2 Evaluation on Synthetic Data

To examine the maximum achievable performance without disk I/O overhead, we load the synthetic files into each client's memory before each test and let the cloud store all post-

Dataset	Raw size	Snapshots	Dedup Ratio	Compress Ratio
DOCKER	70.2 GiB	94	4.2	1.7
LINUX	44.6 GiB	82	2.8	2.3
VM	3.0 TiB	660	33.4	2.0
FSL	3.0 TiB	42	8.2	2.0
MS	3.9 TiB	30	4.1	2.0

Table 1: Characteristics of real-world datasets.

deduplicated data in memory (we include the disk I/O overhead in the evaluation in §6.3). We report the average results over five runs and include the 95% confidence intervals based on student’s t-distribution (except for line graphs).

Exp#1 (Overall performance). We evaluate the upload (download) performance of overall systems. We consider a single client that successively uploads the same 2 GiB file from SYN-Unique *twice*. The client then downloads the same file. We measure the upload (download) speed of each operation. Our goal is to examine the maximum achievable performance of all schemes for storing all non-duplicate data and all duplicate data. Note that the file size is small here, such that all fingerprints can be tracked in the top- k index in DEBE (we consider large-scale datasets in §6.3).

We compare DEBE with three DaE approaches: (i) *DupLESS* [7], which implements server-aided key management based on OPRF (§2.1); (ii) *TED* [49], which generates the key of each chunk based on lightweight hash computations in the key server; and (iii) *CE* [23], the convergent encryption scheme (§2.1). To study the security overhead of DEBE, we include plain deduplication (*Plain*), in which the client uploads the plaintext chunks to the cloud for deduplication and compression through a communication channel protected by SSL/TLS. Unlike DEBE and Plain, the DaE schemes (i.e., DupLESS, TED, and CE) do not realize compression due to incompatibility (§2.1). For fair comparisons, we re-implement all baselines based on their original papers under the same implementation setting (§5) in C++.

Figure 5(a) shows the upload speeds. DEBE outperforms all DaE schemes. When uploading non-duplicate data, DEBE achieves 10.09 \times , 1.42 \times , and 1.25 \times speedups over DupLESS, TED, and CE, respectively, by avoiding the generation of chunk-based keys (note that DupLESS has very low upload speeds due to the expensive OPRF operations). Even though DEBE applies compression, its compression overhead is masked by the performance gain over the key generation overhead of DaE. When uploading duplicate data, DEBE becomes more efficient. Its speedups increase to 13.08 \times , 1.88 \times , and 1.65 \times over DupLESS, TED, and CE, respectively, since it avoids performing encryption and compression on the duplicate chunks. Compared with plain deduplication, DEBE only incurs 21.6% and 7.4% performance overhead for the uploads of non-duplicate and duplicate data, respectively.

Figure 5(b) shows the download speeds. All DaE schemes follow the same download paradigm, in which the client retrieves both ciphertext chunks and encrypted chunk-based keys from the cloud, decrypts each key and the corresponding

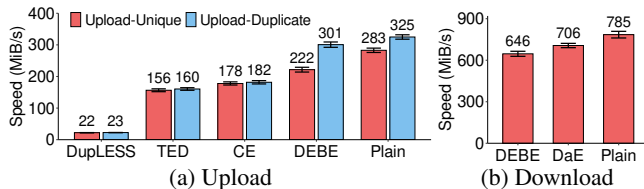


Figure 5: (Exp#1) Overall performance.

Steps	1st upload	2nd upload
Chunking	0.61 \pm 0.01 ms	
Transmission Protection	0.37 \pm 0.01 ms	
Fingerprinting	2.27 \pm 0.04 ms	
Frequency tracking	0.06 \pm 0.01 ms	
First-phase dedup	0.10 \pm 0.01 ms	0.14 \pm 0.01 ms
Second-phase dedup	0.80 \pm 0.02 ms	-
Compression	0.67 \pm 0.01 ms	-
Encryption	0.33 \pm 0.01 ms	-

Table 2: (Exp#2) Breakdown of computational time per processing 1 MiB data in two successive uploads.

chunk, and reconstructs the original file. Compared with DaE, DEBE incurs 8.5% download speed drop due to the OCalls for moving chunks into the enclave for decryption and decompression (§4.5). Also, DEBE and DaE have 17.7% and 10.1% download speed drops compared with Plain, respectively, since they perform decryption on retrieved chunks.

Exp#2 (Upload breakdown). We study the breakdown of the upload performance. We consider the same scenario as Exp#1 (i.e., a client successively uploads the same 2 GiB file from SYN-Unique twice) and measure the computational time of the client and the enclave in different steps in uploads: (i) *chunking*, in which the client partitions the input file into plaintext chunks; (ii) *transmission protection*, in which the enclave exchanges a session key with the client and decrypts received ciphertext chunks; (iii) *fingerprinting*, in which the enclave computes the fingerprint of each plaintext chunk; (iv) *frequency tracking*, in which the enclave estimates the frequency of each plaintext chunk via the CM-Sketch; (v) *first-phase deduplication*, in which the enclave removes duplicate plaintext chunks via the top- k index; (vi) *second-phase deduplication*, in which the enclave queries the full index via OCalls to remove remaining duplicates; (vii) *compression*, in which the enclave compresses the non-duplicate chunks; and (viii) *encryption*, in which the enclave encrypts the compressed chunks with the data key.

Table 2 shows the results (measured by the computational time per 1 MiB of uploads). In the first upload (i.e., uploading non-duplicate data), fingerprinting is the most time-consuming step since it performs expensive computations on all chunks. On the other hand, frequency-based deduplication (including frequency tracking plus first-phase and second-phase deduplication) takes only 18.4% of the overall time. Note that since the storage is empty before the upload, each non-duplicate chunk is treated as a frequent chunk and examined by both the first-phase and second-phase deduplication.

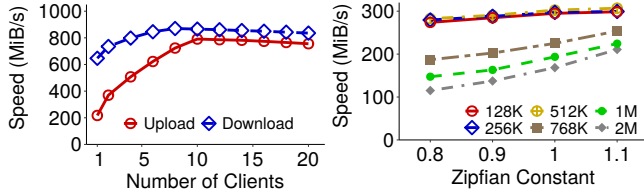


Figure 6: (Exp#3) Multi-client uploads and downloads. **Figure 7:** (Exp#4) Impact of frequency distribution.

In the second upload (i.e., uploading duplicate data), all duplicate chunks are removed by the first-phase deduplication and hence the second-phase deduplication is skipped. In this case, frequency-based deduplication takes only 5.8% of the overall upload time.

Exp#3 (Multi-client uploads and downloads). We evaluate DEBE when multiple clients issue upload/download requests concurrently. In addition to the cloud, we deploy 10 machines, with two client instances each, so as to simulate the concurrent uploads/downloads by up to 20 clients. Each client uploads a 2 GiB synthetic file from SYN-Unique to the cloud, and then downloads the same 2 GiB file. We measure the *aggregate* upload (download) speed as the ratio of the total uploaded (downloaded) data size to the total time all clients complete the uploads (downloads).

Figure 6 shows the results versus the number of clients. The aggregate upload speed first increases with the number of clients and reaches 791.1 MiB/s for 10 clients, followed by dropping to 755.8 MiB/s for 20 clients due to the resource contention in the enclave. The aggregate download speed has a similar trend, and first increases to 870.0 MiB/s and finally drops to 835.7 MiB/s.

Exp#4 (Impact of frequency distribution). We evaluate DEBE on processing the chunks from different frequency distributions. We configure a single client to upload each original chunk of SYN-Freq without chunking, and measure the *computational speed* of the enclave (i.e., including the steps of Table 2 except chunking).

Figure 7 shows the results for different k in the top- k index versus the Zipfian constant. A larger k implies lower performance for all Zipfian constants, since SGX incurs significant paging overhead when the size of enclave contents is greater than 64 MiB [45]. For example, when the Zipfian constant is 0.8, the computational speeds for $k = 512$ K and $k = 1$ M are 282.5 MiB/s and 147.3 MiB/s, respectively. In addition, the computational speed of the enclave increases in more skewed distribution (i.e., a larger Zipfian constant), since the most frequent chunks contribute more duplicates. This mitigates the OCall overhead of querying the full index.

6.3 Evaluation on Real-world Traces

Exp#5 (Performance of deduplication approaches). DEBE’s key design is frequency-based deduplication, and we compare it with other design alternatives. We consider two state-of-the-art memory-efficient deduplication approaches,

namely *similarity-based deduplication* [9] and *locality-based deduplication* [52]. Both approaches store a small in-enclave fingerprint index based on the *feature* of each segment of chunks and perform deduplication by loading a portion of the full index (outside the enclave) into the enclave based on the matched feature. Similarity-based deduplication derives the feature based on the minimum chunk fingerprint of each segment of chunks, while locality-based deduplication generates it by sampling a few fingerprints. As in [9, 52], we choose the segment size as 10 MiB, and the sampling rate of locality-based deduplication as 1/64. While both approaches aim to mitigate disk I/O in plain deduplication, our idea is that they can also be applied to reduce EPC usage, but can only support near-exact deduplication (§4.4).

In addition to the above near-exact deduplication approaches, we include the naïve but exact deduplication baselines for comparisons. Specifically, *in-enclave deduplication* attempts to manage the full index in the enclave; when the full index increases in size and cannot fit into the EPC, it triggers page swapping to evict unused EPC pages to memory. *Out-enclave deduplication* manages the full index in memory, and detects duplicates by issuing OCalls to the full index. For fair comparisons, we include compression over non-duplicate chunks into all baseline approaches. We upload the snapshots of each real-world backup dataset (§6.1) in the order of their creation times. We measure the computational speed of the enclave as in Exp#4.

Figure 8 shows the results. DEBE generally outperforms all approaches. For example, in FSL, it achieves 1.17 \times , 1.20 \times , 1.25 \times , and 2.76 \times average speedups over the similarity-based, locality-based, out-enclave, and in-enclave approaches, respectively. The reason is that DEBE avoids the extra computational overhead of compressing and encrypting some duplicate chunks in both similarity-based and locality-based approaches (which perform near-exact deduplication). Also, it performs the first-phase deduplication and filters out many queries to the full index, thereby mitigating the OCall overhead of the out-enclave deduplication. Although in-enclave deduplication outperforms DEBE when the workload size is small (e.g., the first few snapshots in DOCKER and LINUX), its performance drops dramatically in subsequent snapshots due to expensive paging overhead. DEBE manages lightweight data structures (a CM-Sketch and the small top- k index) in the enclave and mitigates the paging overhead.

Exp#6 (Trace-driven upload and download). Unlike in Exp#1, we evaluate the upload and download performance of DEBE based on real-world data. We enable cloud-side disk I/O, upload all snapshots of each dataset, and finally let the client download them on disk. Here, we only compare DEBE with CE, which is the fastest DaE approach. Since FSL, VM, and MS only include the compressible chunks (§6.1), we let the client machine directly upload chunks without chunking.

Figure 9 shows the speeds for uploading and downloading each snapshot in DEBE and CE. The upload speed of

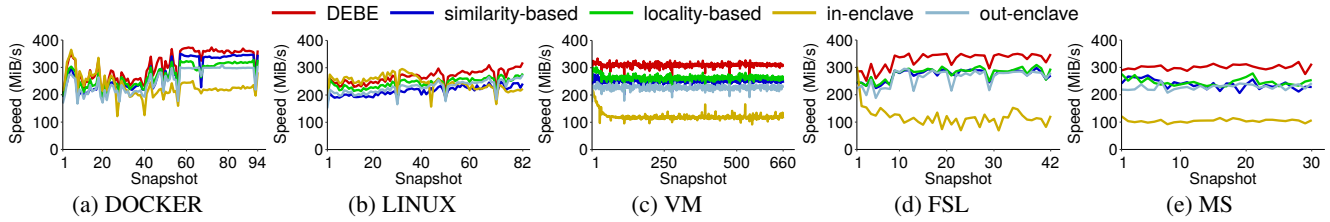


Figure 8: (Exp#5) Performance comparison of different deduplication approaches.

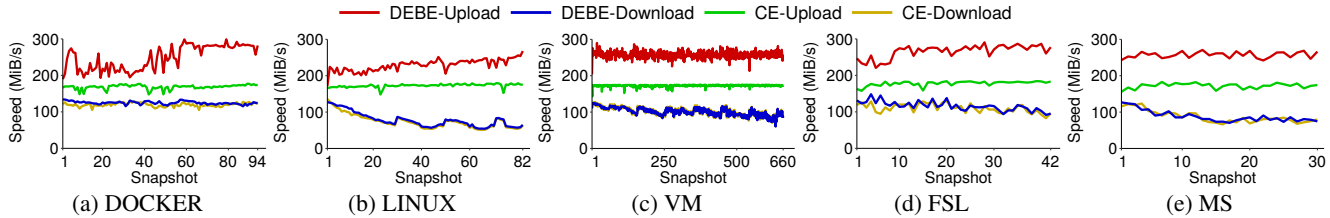


Figure 9: (Exp#6) Trace-driven upload and download performance.

DEBE gradually increases in subsequent snapshots, which include more duplicate plaintext chunks, so DEBE does not need to perform compression and encryption on the duplicate plaintext chunks (removed by deduplication). In contrast, CE is consistently slower than DEBE in uploads, as it performs key generation and encryption for all duplicate plaintext chunks. For example, in FSL, the upload speed of DEBE is 246.5 MiB/s for the first snapshot, and reaches 277.5 MiB/s for the last snapshot. In contrast, the upload speed of CE is 163.5-179.1 MiB/s across all snapshots.

The download speeds of both DEBE and CE are almost the same, since they are throttled by disk I/O. Also, their download speeds decrease across snapshots due to chunk fragmentation [51] (i.e., the chunks of subsequent snapshots become scattered after deduplication), which increases I/O overhead. For example, the download speed of DEBE in FSL is 131.4 MiB/s for the first snapshot, and drops to 95.1 MiB/s for the last snapshot (the download speed of CE is almost the same). Chunk fragmentation can be mitigated via existing approaches [11, 12, 31, 51, 86] and we pose the integration of such approaches into DEBE as future work.

7 Related Work

DaE approaches. Several approaches realize secure deduplication via DaE. In addition to those described in §2.1, some approaches are designed from the security perspectives. Random MLE [1] and iMLE [6] apply non-deterministic encryption to prevent frequency leakage, but they use expensive primitives (e.g., non-interactive zero-knowledge proofs [1], fully homomorphic encryption [6]) that are not ready to be implemented. Liu *et al.* [55] propose to share keys via a decentralized agreement protocol without relying on a dedicated key server, but it introduces expensive performance overhead of interactions among different clients. TED [49] mitigates frequency leakage with a configurable storage blowup. In contrast, DEBE realizes DbE to address both key management overhead and security issues simultaneously.

SGX meets secure deduplication. SGX has been used in secure deduplication. Dang *et al.* [20] employ SGX as a trusted proxy to save network bandwidth for secure deduplication. SPEED [19] performs deduplication for computations inside the enclave to improve resource utilization. You *et al.* [82] leverage SGX to verify the ownership of deduplicated data for secure deduplication. SeGShare [32] builds on a server-side enclave for file-based deduplication, but does not consider fingerprint indexing for chunk-based deduplication. S2Dedup [60] uses a server-side enclave to eliminate a trusted key server for key generation, and it performs deduplication outside the enclave via re-encrypting chunks; in contrast, DEBE directly performs deduplication inside the enclave to protect plaintext chunks. SGXDedup [70] leverages SGX to improve the performance of client-side secure deduplication under DaE. Note that the above SGX-based deduplication approaches are still based on DaE.

8 Conclusion

DEBE realizes an unexplored paradigm, deduplication-before-encryption (DbE), for secure deduplicated storage. It builds on SGX and applies frequency-based deduplication to manage a small fingerprint index for most frequent chunks in the enclave. We show that DEBE outperforms conventional deduplication-after-encryption (DaE) approaches in performance, storage savings, and security.

Acknowledgments

We thank our reviewers and shepherd for their comments. This work was supported in part by the National Natural Science Foundation of China (61972073), the Key Research Funds of Sichuan Province (2021YFG0167, 2020YFG0298), the Sichuan Science and Technology Program (2020JDTD0007), the Fundamental Research Funds for Chinese Central Universities (ZYGX2020ZB027, ZYGX2021J018), CUHK Direct Grant (4055148), and the Research Matching Grant Scheme.

References

- [1] M. Abadi, D. Boneh, I. Mironov, A. Raghunathan, and G. Segev. Message-locked encryption for lock-dependent messages. In *Proc. of CRYPTO*, 2013.
- [2] Advanced Micro Devices Inc. AMD Secure Encrypted Virtualization (SEV). <https://developer.amd.com/sev/>.
- [3] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of USENIX OSDI*, 2002.
- [4] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proc. of ACM HASP*, 2013.
- [5] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O’keeffe, M. L. Stillwell, et al. SCONE: Secure Linux containers with Intel SGX. In *Proc. of USENIX OSDI*, 2016.
- [6] M. Bellare and S. Keelveedhi. Interactive message-locked encryption and secure deduplication. In *Proc. of PKC*, 2015.
- [7] M. Bellare, S. Keelveedhi, and T. Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In *Proc. of USENIX Security*, 2013.
- [8] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In *Proc. of EuroCrypto*, 2013.
- [9] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proc. of IEEE MASCOTS*, 2009.
- [10] J. Black. Compare-by-hash: A reasoned analysis. In *Proc. of USENIX ATC*, 2006.
- [11] Z. Cao, S. Liu, F. Wu, G. Wang, B. Li, and D. H. Du. Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance. In *Proc. of USENIX FAST*, 2019.
- [12] Z. Cao, H. Wen, F. Wu, and D. H. Du. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *Proc. of USENIX FAST*, 2018.
- [13] D. Chen, M. Factor, D. Harnik, R. Kat, and E. Tsfadia. Length preserving compression: Marrying encryption with compression. In *Proc. of ACM SYSTOR*, 2021.
- [14] Cohesity Inc. Cohesity. <https://www.cohesity.com/>.
- [15] Y. Collet. LZ4: Extremely fast compression algorithm. <https://lz4.github.io/lz4/>.
- [16] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
- [17] Couchbase Inc. Couchbase: The modern database for enterprise applications. <https://www.couchbase.com>.
- [18] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. of USENIX OSDI*, 2002.
- [19] H. Cui, H. Duan, Z. Qin, C. Wang, and Y. Zhou. SPEED: Accelerating enclave applications via secure deduplication. In *Proc. of IEEE ICDCS*, 2019.
- [20] H. Dang and E.-C. Chang. Privacy-preserving data deduplication on trusted processors. In *Proc. of IEEE CLOUD*, 2017.
- [21] T. Dinh Ngoc, B. Bui, S. Bitchebe, A. Tchana, V. Schiavoni, P. Felber, and D. Hagimont. Everything you should know about Intel SGX performance on virtualized systems. In *Proc. of ACM SIGMETRICS*, 2019.
- [22] Docker Inc. Docker Hub. <https://hub.docker.com/>.
- [23] J. R. Douceur, A. Adya, W. J. Bolosky, P. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proc. of IEEE ICDCS*, 2002.
- [24] Dropbox Inc. Dropbox. <https://www.dropbox.com/>.
- [25] Druva Inc. Druva. <https://www.druva.com/>.
- [26] A. Duggal, F. Jenkins, P. Shilane, R. Chinthekindi, R. Shah, and M. Kamat. Data domain cloud tier: Backup here, backup there, deduplicated everywhere! In *Proc. of USENIX ATC*, 2019.
- [27] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta. Primary data deduplication-large scale study and system design. In *Proc. of USENIX ATC*, 2012.
- [28] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen. Scalable memory protection in the PENGLAI enclave. In *Proc. of USENIX OSDI*, 2021.
- [29] File System and Storage Lab at Stony Brook University. Traces and snapshots public archive. <http://tracer.filesystems.org>.
- [30] K. Fu, S. Kamara, and T. Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *Proc. of NDSS*, 2006.
- [31] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *Proc. of USENIX ATC*, 2014.

- [32] B. Fuhry, L. Hirschhoff, S. Koesnadi, and F. Kerschbaum. SeGShare: Secure group file sharing in the cloud using enclaves. In *Proc. of IEEE/IFIP DSN*, 2020.
- [33] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of computer and system sciences*, 1984.
- [34] R. Gracia-Tinedo, D. Harnik, D. Naor, D. Sotnikov, S. Toledo, and A. Zuck. SDGen: Mimicking datasets for content generation in storage benchmarks. In *Proc. of USENIX FAST*, 2015.
- [35] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.
- [36] D. Harnik, E. Tsfadia, D. Chen, and R. Kat. Securing the storage data path with SGX enclaves. <https://arxiv.org/abs/1806.10883>, 2018.
- [37] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuivillo. Using innovative instructions to create trustworthy software solutions. In *Proc. of ACM HASP*, 2013.
- [38] J. Ibsen. LZ data generator. <https://github.com/jibsen/lzdatagen>.
- [39] Intel Corporation. Intel(R) Advanced Encryption Standard Instructions (AES-NI). <https://software.intel.com/content/www/us/en/develop/articles/intel-advanced-encryption-standard-instructions-aes-ni.html>.
- [40] Intel Corporation. Intel(R) SHA Extensions. <https://software.intel.com/content/www/us/en/develop/articles/intel-sha-extensions.html>.
- [41] Intel Corporation. Intel(R) Software Guard Extensions. <https://software.intel.com/content/www/us/en/develop/documentation/sgx-developer-guide/top.html>.
- [42] Intel Corporation. Intel(R) Software Guard Extensions SDK for Linux. <https://01.org/intel-softwareguard-extensions>.
- [43] Intel Corporation. Intel(R) Software Guard Extensions SSL. <https://github.com/intel/intel-sgx-ssl>.
- [44] Intel Corporation. Supporting Intel SGX on multi-socket platforms. <https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions/supporting-sgx-on-multi-socket-platforms.html>.
- [45] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh. ShieldStore: Shielded in-memory key-value storage with SGX. In *Proc. of ACM EuroSys*, 2019.
- [46] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *Proc. of USENIX ATC*, 2014.
- [47] J. Li, P. P. C. Lee, Y. Ren, and X. Zhang. Metadedup: Deduplicating metadata in encrypted deduplication via indirection. In *Proc. of IEEE MSST*, 2019.
- [48] J. Li, P. P. C. Lee, C. Tan, C. Qin, and X. Zhang. Information leakage in encrypted deduplication via frequency analysis: Attacks and defenses. *ACM Trans. on Storage*, 16(1):1–30, 2020.
- [49] J. Li, Z. Yang, Y. Ren, P. P. C. Lee, and X. Zhang. Balancing storage efficiency and data confidentiality with tunable encrypted deduplication. In *Proc. of ACM EuroSys*, 2020.
- [50] M. Li, C. Qin, and P. P. C. Lee. CDStore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. In *Proc. of USENIX ATC*, 2015.
- [51] M. Lillibridge, K. Eshghi, and D. Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proc. of USENIX FAST*, 2013.
- [52] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proc. of USENIX FAST*, 2009.
- [53] X. Lin, F. Douglis, J. Li, X. Li, R. Ricci, S. Smaldone, and G. Wallace. Metadata considered harmful ... to deduplication. In *Proc. of USENIX HotStorage*, 2015.
- [54] Linux Foundation. The Linux kernel archives. <https://www.kernel.org/>.
- [55] J. Liu, N. Asokan, and B. Pinkas. Secure deduplication of encrypted data without additional independent servers. In *Proc. of ACM CCS*, 2015.
- [56] U. Maurer. Modelling a public-key infrastructure. In *Proc. of ESORICS*, 1996.
- [57] M. Meehan. Data privacy will be the most important issue in the next decade. <https://www.forbes.com/sites/marymeehan/2019/11/26/data-privacy-will-be-the-most-important-issue-in-the-next-decade/>.
- [58] Memopal. Memopal. <https://www.memopal.com/>.
- [59] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proc. of USENIX FAST*, 2011.
- [60] M. Miranda, T. Esteves, B. Portela, and J. Paulo. S2Dedup: SGX-enabled secure deduplication. In *Proc. of ACM SYSTOR*, 2021.
- [61] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. on Database Systems*, 17(1):94–162, 1992.

- [62] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *Proc. of USENIX Security*, 2011.
- [63] M. Naor and O. Reingold. Number-theoretic constructions of efficient pseudo-random functions. *Journal of the ACM*, 51(2):231–262, 2004.
- [64] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *Proc. of USENIX ATC*, 2018.
- [65] OpenSSL. Cryptography and SSL/TLS toolkit. <https://www.openssl.org/>.
- [66] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: Exitless OS services for SGX enclaves. In *Proc. of ACM EuroSys*, 2017.
- [67] S. Pinto and N. Santos. Demystifying ARM TrustZone: A comprehensive survey. *ACM Computing Surveys*, 51(6):1–36, 2019.
- [68] R. A. Popa, C. M. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. of SOSP*, 2011.
- [69] C. Qin, J. Li, and P. P. C. Lee. The design and implementation of a rekeying-aware encrypted deduplication storage system. *ACM Trans. on Storage*, 13(1):1–30, 2017.
- [70] Y. Ren, J. Li, Z. Yang, P. P. C. Lee, and X. Zhang. Accelerating encrypted deduplication via SGX. In *Proc. of USENIX ATC*, 2021.
- [71] Seagate Technology LLC. Data Age 2025. <https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-data-age-whitepaper.pdf>.
- [72] P. Shah and W. So. Lamassu: Storage-efficient host-side encryption. In *Proc. of USENIX ATC*, 2015.
- [73] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proc. of USENIX FAST*, 2012.
- [74] M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller. Secure data deduplication. In *Proc. of ACM StorageSS*, 2008.
- [75] Z. Sun, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, et al. A long-term user-centric analysis of deduplication patterns. In *Proc. of IEEE MSST*, 2016.
- [76] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proc. of USENIX Security*, 2018.
- [77] G. Wallace, F. Douglass, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proc. of USENIX FAST*, 2012.
- [78] O. Weisse, V. Bertacco, and T. Austin. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. In *Proc. of ACM ISCA*, 2017.
- [79] Z. Wilcox-O’Hearn and B. Warner. Tahoe: The least-authority filesystem. In *Proc. of ACM StorageSS*, 2008.
- [80] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *Proc. of USENIX ATC*, 2016.
- [81] Z. Yang, J. Li, and P. P. C. Lee. Secure and lightweight deduplicated storage via shielded deduplication-before-encryption. Technical report, The Chinese University of Hong Kong, 2022. <http://www.cse.cuhk.edu.hk/~pcllee/www/pubs/tech.debe.pdf>.
- [82] W. You and B. Chen. Proofs of ownership on encrypted cloud data via Intel SGX. In *Proc. of ACNS*, 2020.
- [83] W. Zhang, D. Agun, T. Yang, R. Wolski, and H. Tang. VM-centric snapshot deduplication for cloud data backup. In *Proc. of IEEE MSST*, 2015.
- [84] W. Zhang, H. Tang, H. Jiang, T. Yang, X. Li, and Y. Zeng. Multi-level selective deduplication for VM snapshots in cloud storage. In *Proc. of IEEE CLOUD*, 2012.
- [85] B. Zhu, K. Li, and R. H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. of USENIX FAST*, 2008.
- [86] X. Zou, J. Yuan, P. Shilane, W. Xia, H. Zhang, and X. Wang. The dilemma between deduplication and locality: Can both be achieved? In *Proc. of USENIX FAST*, 2021.

A Artifact Appendix

Abstract

Our artifact consists of the prototypes of DEBE and all baseline approaches, a trace analysis tool for frequency leakage measurement, and the scripts to run all our experiments in §6. The DEBE prototype is a shielded DbE-based deduplicated storage system that supports secure deduplication via Intel SGX. It supports upload/download operations to allow multiple clients to securely outsource their data storage to the cloud. It applies frequency-based deduplication and implements the designs described in §4.

Scope

Our artifact can be used to validate our main claim that DEBE outperforms conventional DaE approaches in performance, storage efficiency, and security. Specifically, our artifact can be used to validate the results shown in the figures and tables

in §6 to support our main claim. In addition, our artifact can be used to run the workloads independent of our evaluation in §6.

Contents

The artifact comprises the following sub-directories:

- `./Prototype`, which includes the implementation of the DEBE prototype.
- `./Baseline`, which includes the implementation of all baseline approaches (e.g., DupLESS, TED, CE, and Plain) used in Exp#1 and Exp#6.
- `./Sim`, which includes a trace analysis tool to measure frequency leakage of CE, TED, and DEBE (see Exp#9 in our technical report [81]).

Also, each sub-directory has a separate README file to introduce the build instructions and usage.

Hosting

Our artifact is available on GitHub. Users can obtain the artifact from the repository <https://github.com/yzr95924/DEBE>. The version we provided for the artifact evaluation is marked with the `atc22ae` tag. We encourage the users to use the latest version of the repository, since it may include bug fixes. We also release the traces used in §6. The README file (<https://github.com/yzr95924/DEBE/blob/master/README.md>) describes the detailed instructions to collect the traces.

Requirements

We developed and evaluated our artifact on a local cluster of 11 machines connected with 10 GbE. Each machine has a quad-core 3.4 GHz Intel Core i5-7500 CPU and 32 GiB RAM running Ubuntu 16.04. We implement DEBE based on Intel SGX SDK Linux 2.7 [42], OpenSSL 1.1.1 [65], and Intel SGX SSL 1.1.1 [43]. The DEBE prototype is written in C++ and compiled by Clang 3.8.0. To validate the basic upload/download operations of DEBE, users need to prepare at least two machines, one of which needs to support Intel SGX to run as the cloud. We recommend users to check the SGX-supported device in <https://github.com/ayeks/SGX-hardware>.

Note that if the user's OS version is higher than Ubuntu 16.04 LTS (e.g., Ubuntu 20.04 LTS), it might not be able to install the packages with the same versions as in our paper. Nevertheless, we expect that the impact of using the packages with newer versions would be limited and our prototype can still run correctly.

Workflow

To reproduce the experiments in §6, users can refer to `./Prototype/ae_instruction.md` for the detailed instructions.

RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing

^{1,2}Zijun Li, ¹Jiagan Cheng, ¹Quan Chen, ²Eryu Guan, ²Zizheng Bian, ²Yi Tao, ²Bin Zha, ²Qiang Wang, ²Weidong Han, ¹Minyi Guo
¹Department of Computer Science and Engineering, Shanghai Jiao Tong University
²Alibaba Group

Abstract

The secure container that hosts a single container in a micro virtual machine (VM) is now used in serverless computing, as the containers are isolated through the microVMs. There are high demands on the high-density container deployment and high-concurrency container startup to improve both the resource utilization and user experience, as user functions are fine-grained in serverless platforms. Our investigation shows that the entire software stacks, containing the cgroups in the host operating system, the guest operating system, and the container *rootfs* for the function workload, together result in low deployment density and slow startup performance at high-concurrency. We propose and implement a lightweight secure container runtime, named **RunD**, to resolve the above problems through a holistic guest-to-host solution. With RunD, over 200 secure containers can be started in a second, and over 2,500 secure containers can be deployed on a node with 384GB of memory. RunD is adopted as Alibaba serverless container runtime to support high-density deployment and high-concurrency startup.

1 Introduction

With serverless computing (Function-as-a-Service), tenants submit functions directly to the Cloud without renting virtual machines, and the cloud provider uses containers to host invocations on-demand [26, 31, 35, 48, 48, 56]. Most cloud providers publish the serverless computing services with the pay-for-use pricing model, such as Amazon Lambda [4], Google Cloud Function [11], Microsoft Azure Functions [13], and Alibaba Function Compute [2].

When hosting function invocations, traditional containers (e.g., Docker, LXC) only provide process level isolation [22, 38], as they are implemented based on *Namespace* and *Cgroup*. They cannot prevent privilege escalation, information disclosure side channels, and covert channel communication [20]. To this end, secure containers that achieve the same isolation with the traditional virtual machines are

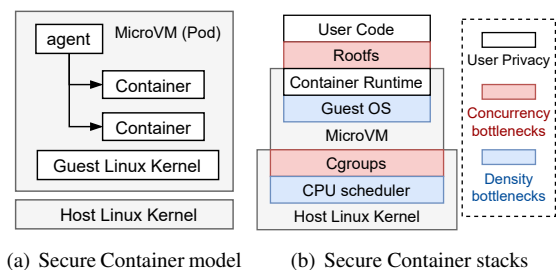


Figure 1: The state-of-the-art secure container model, and several bottlenecks in the architecture stacks.

often preferred. MicroVM is for isolation, and the container is for abstraction [30]. Secure container often creates a normal container within the lightweight microVM as shown in Figure 1(a). In such way users can build serverless services based on existing container infrastructure and ecosystem. It ensures compatibility with the container runtime in the MicroVM. Kata Containers [19] and FireCracker [20] provide practical experience in implementing such secure containers.

Figure 1(b) shows the architecture hierarchy of a secure container. In general, the guest operating system (GuestOS) in the microVM and resource scheduling on the host are off-loaded to the cloud provider. The *rootfs* is a filesystem and acts as the execution environment of user code. It is created by the host and passed to the container runtime in the microVM. On the host side, cgroups are used to allocate resources to secure containers, and the CPU scheduler manages the resource allocation. The complex hierarchy of secure containers brings extra overhead.

The lightweight and short-term features of functions make high-density container deployment and high-concurrency container startup essential for serverless computing. For instance, 47% of Lambdas run with the minimum memory specification of 128MB [5] in AWS, about 90% of the applications never consume more than 400MB in Microsoft Azure [49]. Since a physical node often has large memory space (e.g., 384GB), it should be able to host many func-

tions. Meanwhile, a large number of function invocations may arrive in a short time. However, the overhead of secure containers significantly reduces the deployment density of functions, and the concurrency of starting containers.

Our investigation identifies two key factors in secure containers that result in low concurrent startup. First of all, *rootfs* either results in unacceptable long latency for writable device provisioning or high CPU overhead under considerable I/O stress, when many containers are started concurrently. Secondly, concurrently starting multiple containers brings a large number of cgroups operations on the host side. However, the cgroup-related operations are serialized in the operating systems. The serialization is due to several mutex locks introduced in the kernel to handle a complex hierarchy of cgroup subsystems. The serial operations slow down cgroups creation for microVMs.

Meanwhile, secure containers amplify the resource overhead of each function, multiply host-side resource consumption with more microVMs, and lower the deployment density. Firstly, for microVMs, the standard Linux kernel is heavyweight for a small-sized memory specification. Secondly, the mainstream block-based solution for container *rootfs* in microVM generates the same page cache in both host and guest, resulting in a duplicated memory overhead. Lastly, CFS (Completely Fair Scheduler) in the host operating system traverses all the cgroups (containers) for balancing the processes, resulting in a significant scheduling overhead at high-density deployment.

We propose and implement a lightweight secure container runtime, named **RunD**, to resolve the above problems through a holistic guest-to-host solution. According to our evaluation, RunD boots to application code in *88ms*, and can launch 200 secure containers per second on a node. On a node with 384GB memory, over 2,500 secure containers can be deployed with RunD.

The main contributions of this paper are as follows.

1. **Bottlenecks identification in high-density deployment and high-concurrency startup of secure containers in serverless.** We analyze the shortcomings and bottlenecks through a holistic guest-to-host solution, in terms of container *rootfs* storage, the microVM memory footprint, and the overhead of cgroups.
2. **A guest-to-host solution to secure containers for high-density and high-concurrency targets in serverless.** The practice including: 1) a better container *rootfs* implementation based on read/write splitting for serverless; 2) the method to condense the guest kernel and improve kernel sharing by a pre-patched kernel image; 3) the host-side lightweight cgroup design and the rename-based cgroup pool management.
3. **A lightweight serverless runtime RunD for serverless architecture.** We design and open-source RunD based on Kata-runtime, and it shows much higher de-

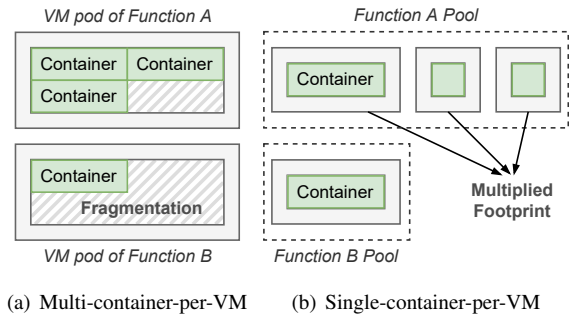


Figure 2: Two practices of the secure container model.

ployment density and startup concurrency compared with the state-of-the-arts.

RunD is adopted as Alibaba serverless container runtime serving more than 1 million functions and almost 4 billion invocations daily. The online statistics demonstrate that RunD enables the maximum deployment density of over 2,000 containers per node and supports booting at most 200 containers concurrently with a quick end-to-end response.

2 Background

In this section, we will discuss the current secure container design, and concerning problems motivating this work.

2.1 Secure Container Models

Based on different levels of security/isolation requirements, there are generally two categories of secure containers in the production environments.

Figure 2(a) shows the *multi-container-per-VM* secure container model that only isolates functions. In the model, a virtual machine (VM) hosts the containers for the invocations of the same function. The containers in the same VM share the guest operating system of the VM. In this case, the invocations to different functions are isolated, but the invocations to the same function are not isolated. Since the number of required containers for each function varies, this model results in memory fragmentations [34]. Though the memory fragmentations can be reclaimed at runtime, it may significantly affect the function performance, and even crash the VM when the memory hot-unplug fails.

Figure 2(b) shows the *single-container-per-VM* secure container model that isolates each function invocation. Current serverless computing providers [1, 20] mainly use this secure container model. In this model, each invocation is served with a container in a microVM. This model does not introduce memory fragmentations, but the microVMs themselves show heavy memory overhead. It is obvious that each microVM needs to run its exclusive guest operating system, multiplying the memory footprints.

The secure container depends on the security model of hardware virtualization and VMM, explicitly treating the guest kernel as untrusted through syscall inspections. With the prerequisite of isolation and security, this work targets the *single-container-per-VM* secure container model.

2.2 Problems with Secure Containers

In production serverless platforms, achieving high container startup concurrency, and high container deployment density are the two key requirements [20]. With the *single-container-per-VM* secure container model, there are problems in achieving the two purposes.

Requirement on high-concurrency container startup.

In serverless platforms, each function invocation is short, and a large number of function invocations may arrive in a short time. For example, in Alibaba serverless platform, more than 200 container-launch requests arrive nearly simultaneously on a node. The latency until all containers have entered *main()* can swell super-proportionally due to resource contention among the simultaneously launching VMs. Meanwhile, emerging internet services often show a diurnal load pattern and have bursty loads [18]. A large number of containers are required to be created when the load bursts. Some techniques, such as prewarming containers [31, 42, 49], are able to alleviate container cold startups.

However, bursty loads are inevitable can easily exhaust the limited prewarmed containers. The ability to startup containers at high-concurrency is crucial for serverless platforms.

Requirement on high-density container deployment.

The small container specification in a serverless computing platform brings the requirement to deploy containers densely on a node. For instance, 47% of lambda functions run with the minimum memory specification of 128MB in AWS [5]. The actual memory usage of a container may also be smaller than its specifications. As Azure reports [49], about 90% of the applications never consume more than 400MB of memory. A node with 256GB of memory can host $8 \times 256 = 2048$ containers if there is no other overhead. In Alibaba serverless platform, over 2,500 secure containers that 128MB-sized can be deployed on a node with 384GB memory.

Without proactive customizations, secure containers incur extra memory overhead, reducing deployment density in serverless computing. Increasing deployment density greatly improves resource utilization and multi-tenant serving efficiency with the same infrastructure.

3 Problem Analysis and Insights

In this section, we analyze the problems of achieving high-concurrency startup and high-density deployment with secure containers. We use Kata container [19] as the representative secure container to perform the following studies.

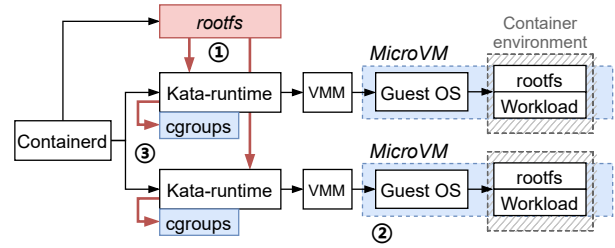


Figure 3: The steps of starting up multiple Kata containers concurrently. The concurrency bottleneck results from creating *rootfs* (step ① in red block) and creating *cgrouops* (step ③ in red flowline). The density bottlenecks result from the memory footprint of the microVM (step ② in blue block) and the scheduling of massive *cgrouops* (step ③ in blue block).

Figure 3 shows the steps of starting Kata containers. First, *containerd* concurrently creates the container runtime *Kata-runtime* and prepares *runc*-container *rootfs*. Second, the hypervisor loads the GuestOS and the prepared *rootfs* to launch a *runc*-container in the microVM. Third, the function workload is downloaded into the container and may start to run.

Comparing with starting traditional containers [53], we have two observations when starting up secure containers.

- When starting 100 or more Kata containers concurrently, there is a distinct performance degradation of creating *rootfs* and *cgrouops*, during the *Kata-runtime* preparation. The degradation results in the low concurrency of starting containers.
- When deploying more than 1,000 Kata containers with 128MB memory specification on a single node with 384GB memory and 104 cores, the microVMs' memory footprint (due to the guest kernel and *rootfs*) already occupies most of the memory space. Meanwhile, the containers' I/O performance is also seriously degraded.

Figure 3 also shows three bottlenecks we found that result in the above two observations. In general, the inefficiency of creating *rootfs* and *cgrouops* results in low container startup concurrency. The high memory footprint and scheduling overhead result in low container deployment density. We analyze the bottlenecks in the following subsections.

3.1 Bottleneck of Container Rootfs Storage

In general, *rootfs* can be exposed to the container runtimes in the microVMs through two interfaces to construct the image layers: filesystem sharing (e.g. *9pfs* [45], *virtio-fs* [16]) and block device (e.g. *virtio-blk* [46]). Filesystem sharing enables microVMs to access a directory tree on the host directly. When the block device is used, the host creates block devices through the device-mapper [8] and passes them to the microVMs, so that containers can access data at the block level, rather than the file level.

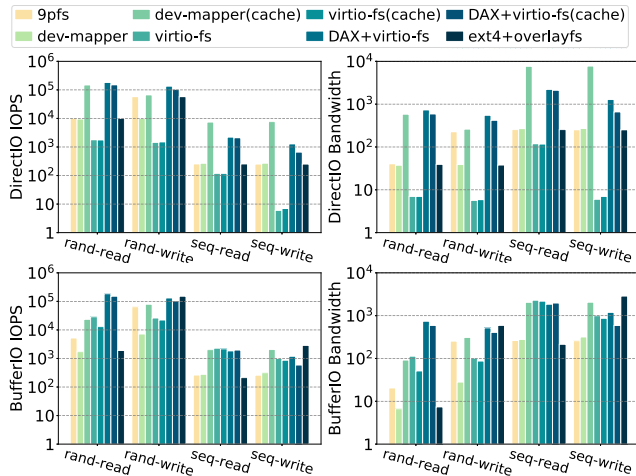


Figure 4: The IOPS/bandwidth performance of rand/seq directIO/BufferIO read/write when using different *rootfs* mapping in Kata-runtime (dev-mapper represents that *virtio-blk* is used, *ext4+overlaysfs* represent the baseline of default runc-container *rootfs* implementation).

Figure 4 shows the IOPS (IO-Per-Second) and IO bandwidth of the random/sequential read and random/sequential write, when Kata uses *9pfs*, *virtio-fs*, and *virtio-blk*, respectively. We also measure the metrics of using *ext4* file system and *overlaysfs* file access interface on the host node, to denote the case of the traditional containers [50, 51]. As observed, microVMs should not use *9pfs* as *rootfs* storage interface due to the poor performance.

With the default configuration (cache enabled), *virtio-blk* performs best at random/sequential writing. However, the device-mapper who prepares the block device in the host cannot meet the high-concurrency requirement [59]. According to our measurement, it takes as high as 10 seconds to prepare a *rootfs* when 200 containers are started concurrently, while it only takes about 30 milliseconds for a single container startup. In this case, the operation of preparing *rootfs* timeouts, resulting in the container breakdown. Moreover, *virtio-blk* inherently does not support the page cache sharing between host and guest operating systems. When *virtio-blk* backend reads *rootfs* files into the host page cache, the mapped content reproduces the same page cache in the guest. The issue of duplicated page cache brings a high memory footprint overhead.

Virtio-fs resolves the problem of duplicating page cache. When *DAX* is enabled in *virtio-fs*, it allows bypassing guest page cache and mapping host page cache directly in guest address space [16]. However, *virtio-fs* results in poor random/sequential write performance (Figure 4). In addition, each container has to employ a client daemon to support *virtio-fs* I/O operations, leading to excessive CPU usage when enormous containers colocate. Things get worse for

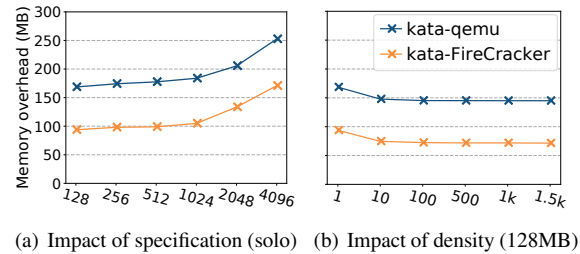


Figure 5: The memory overhead of a secure container.

either large I/O stress under high-concurrency or massive operations of metadata processing.

The above investigation shows that either *virtio-fs* or *virtio-blk* can compromise either deployment density or startup concurrency of secure containers. An exploratory alternative would be: using *virtio-fs* to support the read-only part of *rootfs* for sharing page cache between host and guests, and using *virtio-blk* to support the writeable part of *rootfs* for high I/O performance. A solution is also required to further reduce the duplicated writable part for *rootfs*.

3.2 High Memory Overhead Per Container

Except for the memory used by the user function, the memory footprint of other components in the secure container is the memory overhead. The 5MB memory overhead reported in FireCracker [52] is the overhead of the FireCracker VMM itself. In the microVM of a secure container, the guest operating system, the struct page for memory management, and other components (e.g., baseOS, shimv2, agent) also consume additional memory space [52].

Figure 5 shows the per-container memory overhead of secure containers with different memory specifications and at different deployment densities. In the figure, *Kata-qemu* is the secure container that uses *qemu* as the hypervisor, and *Kata-FireCracker* uses *FireCracker* as the hypervisor. As observed in Figure 5(a), the memory overheads of a 128MB container are 94MB and 168MB with *Kata-FireCracker* and *Kata-qemu*, respectively. The overhead increases with the memory specification of the container.

The average memory footprint of a single microVM can be reduced by sharing the text/rodata segment among multiple microVMs. Mainstream MicroVMs achieve it by mapping the kernel file to the guest memory directly using *mmap*. As shown in Figure 5(b), the per-microVM memory overhead of *kata-qemu* and *kata-FireCracker* reduce to 145MB and 71MB when 1,000 VMs are deployed on a node. However, the overhead is still too large for a serverless container with only 128MB memory specification.

MicroVM template (e.g., Kata template) [17, 29, 52] is a popular method to further reduce the per-microVM memory overhead, while preserving microVM consistency. The

template serves as a primary image for a microVM copy that includes disks, devices, and settings. New microVMs are created by on-demand forking from a pre-created template microVM, and text/rodata segments are also shared among multiple microVMs [54] in read-only mode. The unaccessed kernel files of the template will not consume the physical memory, reducing the memory overhead.

However, the template technique is not as efficient as we thought, due to the self-modifying codes in the operating system kernel [24, 25]. The self-modifying code technique alters the instructions on-demand as it runs, and the Linux kernel relies heavily on self-modification code to improve performance on boot and during runtime. We start a clean microVM with CentOS 4.19 guest kernel from a template to investigate the impact of self-modifying codes. The investigation shows that 10,012KB of the code and the read-only data is accessed in the memory, but 7,928KB of them were modified during boot. This case in point reveals that the self-modifying codes degrade the efficiency when using *mmap* for less memory consumption of kernel image files.

The code self-modifying reduces the shareable memory when using microVM template. Reducing the self-modifying codes in the guest kernel is worth investigating if they are not necessary for the serverless computing scenario.

3.3 High Host-side Overhead of Cgroups

Cgroup is designed for resource control and abstraction of processes. In serverless computing, the frequency of function invocations shows high variation. In this case, the corresponding secure containers are frequently created and recycled. For instance, in our serverless platform, at most 200 containers would be created and recycled on a physical node concurrently in a second. The frequent creating and recycling challenge the cgroup mechanism on the host.

We measure the performance of cgroup operations when creating 2,000 containers concurrently. In the experiment, we use different numbers of threads to perform cgroup operations. Figure 6(a) shows the cumulative distribution of container creating latencies. Counter-intuitively, the latency increases when more threads are used, even if each thread needs to create fewer containers.

The reason behind the above fact is that the Linux kernel introduces several global locks (e.g., *cgroup_mutex*, *css_set_lock*, *freezer_mutex*) to serialize cgroup operations. The global locks are used to coordinate more than 10 resource subsystems (aka. the cgroup subsys) involved in cgroup. Figure 6(b) shows the flame graph of creating 2,000 cgroups using 10 threads concurrently. In the figure, the red parts show the case that “mutex locks” are active. When the cgroup mutex uses the optimistic spinning by default, the spinner cgroups experience the optimistic spinning if they fail to acquire the lock. It will lead to heavy CPU consumption and belated exiting of the critical section in the multi-

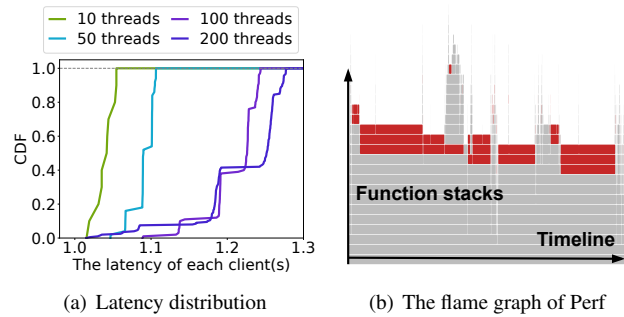


Figure 6: The performance of cgroup operations when creating 2,000 containers concurrently. Due to the mutex lock, the cgroup operations have higher latencies as the concurrency increases.

threaded scenarios. *Therefore, the locks serialize the operations of cgroups and drag down the latencies.*

Besides, a common observation is that there are often more than 10,000 cgroups with thousands of containers on a compute node. The PELT (Per-Entity Load Tracking) for load balancing in CFS will iterate over all cgroups and processes when scheduling these containers. In this scenario, the frequent context switching and hotspot functions that involve high-precision calculation in the scheduler become a bottleneck, accounting for 7.6% of the CPU cycles of the physical node, according to our measurement.

The host-side overhead of cgroups prohibits the high-density deployment and high-concurrency startup in serverless computing. Simplifying the cgroup design, and reducing the critical section introduced by the mutex locks, are fundamental solutions to eliminate the high host-side overhead.

4 Methodology of RunD

The above analysis reveals the bottlenecks in the host, the microVM, and the guest in achieving the high-concurrency startup and high-density deployment. We propose **RunD**, a holistic secure container solution that resolves the problem of duplicated data across containers, high memory footprint per VM, and high host-side cgroup overhead.

In this section, we first show a general design overview of RunD, and then present the design of each component to resolve the corresponding problem.

4.1 Design Overview

When designing RunD, we have a key implication for serverless runtime. The negligible host-side overhead in a traditional VM can cause amplification effects in the FaaS scenario with high-density and high-concurrency, and any trivial optimization can bring significant benefits. Utilizing the

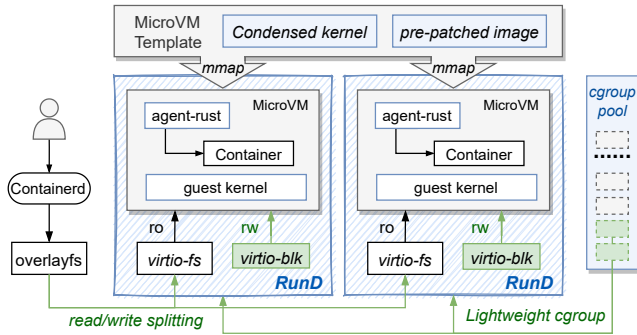


Figure 7: The lightweight serverless runtime of RunD. The condensed kernel and pre-patched image land in the guest domain, while read/write splitting-based *rootfs* and a lightweight cgroup pool land in the host domain.

features of read-only data/runtime and non-persistent storage in serverless, RunD proposes guest-to-host solutions.

Figure 7 shows the RunD design and summarizes our methodologies. RunD runtime makes a read/write splitting by providing the read-only layer to *virtio-fs*, using the built-in storage file to create a volatile writeable layer to *virtio-blk*, and mounting the former and latter as the final container *rootfs* using *overlaysfs*. RunD leverages the microVM template that integrates the condensed kernel and adopts the pre-patched image to create a new microVM, further amortizing the overhead across different microVMs. RunD renames and attaches a lightweight cgroup from the cgroup pool for management when a secure container is created.

Based on the above optimizations, a secure container (referred to as a “sandbox”) is started in the following steps, when RunD is used as the secure container runtime.

- In the first step, once *containerd* receives a user invocation, it forwards the request to RunD runtime.
- Second, RunD prepares the runc-container *rootfs* for the virtual machine hypervisor. The *rootfs* is separated into read-only layer and writable layer. (Section 4.2).
- Third, the hypervisor uses the microVM template to create the required sandbox (Section 4.3), and mount the runc-container *rootfs* into the sandbox by *overlaysfs*.
- Lastly, a lightweight *cgroup* is attached to the sandbox (Section 4.4), to manage the resource allocation for this sandbox in the host.

4.2 Efficient Container Rootfs Mapping

Section 3.1 examines the challenges in the high-density and high-concurrency scenario for container *rootfs*. The current secure container fails to discriminate between serverless platforms and traditional infrastructure-as-a-service environments. The mainstream solutions are designed for persistent

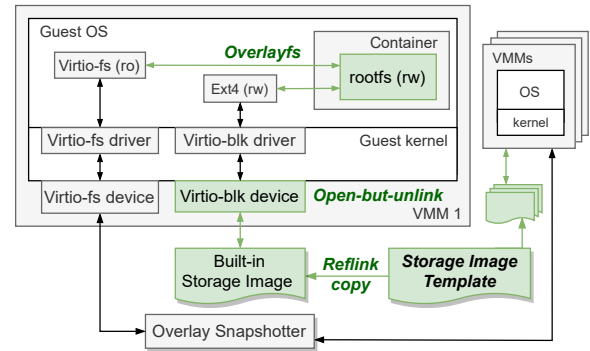


Figure 8: The read/write splitting of container *rootfs*. *Virtio-fs* is used to handle the read-only layer and *virtio-blk* is used to handle the volatile writable layer.

data storage, and it is the key point why container *rootfs* storage imposes restrictions on our goals.

We investigate the data in a sandbox in the serverless computing scenario, and find that **user-provided code/data is read-only for the operating system, and the system-provided runtime files are also read-only for user functions**. Meanwhile, the data in the local memory or storage generated in a sandbox will not be used by subsequent function invocations, due to the stateless feature of serverless computing. The temporary and intermediate data generated during the function execution is not required to be persisted.

Based on the above finding, it is possible to split the *rootfs* into a read-only layer and a writable layer, and then handle them in different ways [32]. The sandboxes can share the read-only layer on the same node, and the writable layer has to be prepared separately for each sandbox.

Figure 8 shows the way to split *rootfs* into a read-only layer and a volatile writable layer. According to the investigation in Section 3.1, *virtio-fs* is used to handle the read-only layer, and *virtio-blk* is used to handle the volatile writable layer for better performance. The read-only layer is stored in the host and can be prepared in negligible time when using the overlay snapshotter provided by the container runtime. However, it is challenging to handle the volatile writable layer efficiently. By default, the host operating system needs to prepare a logic storage volume for the sandbox. This operation is time-consuming and is one of the most important reasons that result in the long latency of preparing *rootfs*.

We propose the volatile block device as the volatile writable layer, considering the volatile feature of the writable layer in serverless platforms. The volatile block device will not persist temporary data from user functions to the disk, unlike the logic storage volume. A *storage image template* is pre-created in the host as the base file. When creating a volatile block device for a new sandbox, a *build-in storage image* is created and linked to the *storage image template*, using *reflink* [60]. *reflink* enables *storage image template* to

share data with *build-in storage images* in a CoW (Copy-on-Write) fashion. Then a volatile block device is created associated with the *build-in storage image*. Once the hypervisor opens the device, the *build-in storage image* will be deleted. The volatile block device ensures that user functions can perform writing as usual without persisting data on the local disk.

We compare our solution with the traditional ones in which the entire *rootfs* is created by the device-mapper as a block device. When 200 sandboxes are started concurrently, traditional solutions incur 4,500 IOPS and use 100MB/s IO bandwidth. On the contrary, our solution incurs only 1,500 IOPS and uses 8MB/s IO bandwidth. Better, the time needed to prepare the *rootfs* decreases from 207ms to only a negligible 0.2ms, and the writing performance of our solution is the same as that of the mainstream transmission.

4.3 Condensed and Pre-patched Guest Kernel

In this subsection, we present two techniques used to reduce the memory used by each sandbox, so that the deployment density can be significantly increased.

4.3.1 Reducing the guest kernel size

Following the abstraction premise in current serverless platforms, the guest environment management for serverless containers is offloaded to the cloud provider. Meanwhile, RunD depends on the security model of hardware virtualization and VMM, explicitly treating the guest kernel as untrusted through syscall inspections. Based on this fact, there is an opportunity to condense the guest kernel for the lightweight characteristic of serverless functions. Considering that several features in the guest kernel are redundant and memory intensive in the serverless context, RunD condenses these features at compile-time. When customizing the condensed guest kernel, the principles behind it are as follows:

- Minimize kernel memory footprint and image size.
- Retain features required in the serverless context.
- Without runtime performance degradation.

Following the above principles, we build the condensed kernel for the guest operating system based on Linux kernel, by disabling features:

- Do not pre-create loop device (2.2MB Mem reduced).
- Disable *acpi* and *ftrace* (2MB and 6MB Mem reduced).
- Disable *graphics*-related items (2MB Mem reduced).
- Disable *i2c* and *ceph* (3MB Mem reduced, and 4MB reduced of kernel image size).
- Kernel files (560K Mem and 571K image size reduced).

Validating all features at compile-time case by case, RunD effectively reduces the memory footprint of a CentOS 4.19 Linux kernel by about 16MB and condenses the kernel image by about 4MB. Based on this condensed guest kernel, we

review several investigations of the self-modifying code and propose our solution to reduce the memory overhead further.

4.3.2 Alleviating code self-modification

As mentioned before, cloud providers manage and maintain the underlying hardware and execution runtimes in serverless context, standing for that all microVMs on the same node generally use the same guest kernel. In this scenario, the sandboxes on the same node generate the same patched kernel code, even if they execute the self-modification patch logic. This is because **the self-modifying code of kernel text segments only occurs at the startup phase, after which the kernel code area becomes “read-only after initialization”**. In this case, sandboxes experience the same initialization phase and generate predictable self-modifying code segments.

Based on the above observation, there is an opportunity to generate a pre-patch guest kernel image file already patched with self-modified code segments. The MicroVM template technique discussed in Section 3.2 may work efficiently without self-modifying code.

Adapting to this optimization, we also resolve the potential kernel panic issues when loading the pre-patched kernel image for higher stability. RunD tries to share as many kernel files as possible across different secure containers. With a pre-patched microVM template, RunD not only reduces the memory footprint of a single container for higher-density deployment, but also allows to quickly fork multiple instances [29, 52].

4.4 Lightweight Cgroup and Cgroup Pool

In Section 3.3, we analyze that serialized cgroups operations in the host become one of the bottlenecks of secure containers with high-density deployment and high-concurrency startup. The intuitions are to efficiently handle synchronization access on mutex structures and reduce the number of cgroups with a better design.

Our further investigations reveal the optimization opportunities in two aspects. Firstly, creating containers involves multiple cgroup subsystems (e.g. *cpu*, *cpuacct*, *cpuset*, *memory*, and *blkio*). Because the Linux kernel cannot parallelize these cgroup-related operations, creating these groups for each sandbox is time-consuming. Secondly, **pre-creating and maintaining cgroups in a pool can effectively reduce the creation overhead, since afterward only the cgroup rename is used**. The cgroup rename, as a special case, is a lightweight operation without acquiring any global lock. Following these two observations, we propose a lightweight cgroup and the cgroup pool, as shown in Figure 9.

The lightweight cgroup decreases the total number of cgroups and system calls. Rather than creating the cgroup for each subsystem, we aggregate necessary cgroup subsystems

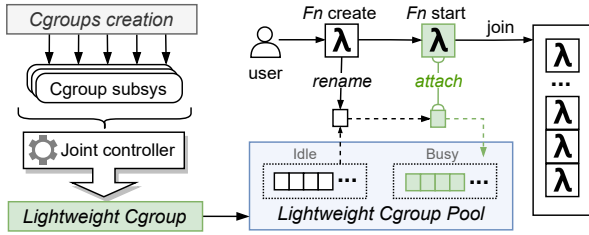


Figure 9: The lightweight Cgroup aggregates all subsystems, eliminating the time-consuming creation by renaming from the Cgroup pool.

(aka the *cpu*, *cpuacct*, *cpuset*, *memory*, and *blkio*) into one single dedicated lightweight cgroup. The implementation of the joint cgroup controller helps RunD reduce the redundant cgroup operations when a container is started, significantly decreasing the total number of cgroups and system calls.

The cgroup pool with renaming mechanism eliminates the time-consuming cgroup creation and initialization. RunD pre-creates corresponding lightweight cgroups and maintains them in a cgroup pool based on the pre-defined node capacity. These cgroups are marked idle when initialized, and are protected in a linked list. For each created container, RunD simply allocates an idle cgroup, updates the state to busy, performs the `cgroup rename` operation, and then attaches the container to this renamed cgroup when a container is started. If a container triggers recycling, RunD will take the cgroup back to the pool, kill the corresponding instance process, and then update the returned cgroup state to idle for subsequent allocating and renaming.

Adopting the above optimizations in kernel mode, we replay the evaluation in Section 3.3. The cgroups creation only consumes 0.09s (1 thread), 0.1s (50 threads), and 0.14s (200 threads), respectively. Compared with the default mechanism, the lightweight cgroup and the rename-based cgroup pool reduce 94% of the cgroups creation time.

5 Evaluation

In this section, we evaluate the performance of RunD in supporting high-concurrency startup and high-density deployment of secure containers, and introduce the performance of RunD in production usage.

5.1 Evaluation Setup

We have implemented and open-sourced RunD with Rust, a more memory-efficient and thread-safe programming language. RunD runtime involves four main modules: Containerd-shim (21k LOC), Device (4.4k LOC), Hypervisor (5.6k LOC), and Lightweight-cgroup (20k LOC).

Table 1: Experiment setup in our evaluation.

Configuration		
Hardware	CPU: 104 vCPUs (Intel Xeon Platinum 8269CY) Memory: 384GB, two SSD drives: 100GB, 500GB	
Software	OS: CentOS7, kernel: Linux kernel 4.19.91	
Container	kata-qemu	containerd 1.3.10, kata 1.12.1
	kata-FC	containerd 1.5.8, kata 2.2.3
	kata-template	containerd 1.3.10, kata 1.12.1
	RunD	containerd 1.3.10

Baselines: we compare RunD with the state-of-the-art secure container, Kata Containers [19]. Specifically, we use three popular configurations of Kata containers: *Kata-qemu*, *Kata-template*, and *Kata-FC*. *Kata-qemu* uses QEMU [15, 23] as the microVM hypervisor, *Kata-template* uses QEMU while integrating container template, *Kata-FC* uses lightweight FireCracker [20] as the microVM hypervisor. *Kata-qemu* and *kata-template* use an old version of Kata Containers, as the new version has some bugs that result in poor performance. Table 1 shows the detailed setups.

Testbed: we run the experiments on a node with 104 virtual cores, 384GB memory, and two SSD drives of 100GB and 500GB. Such specification is widely-used in production clouds. The 100GB drive is used as the root filesystem of the host operating system, and the 500GB drive is used by the secure containers. We use Alibaba Cloud Linux 2 for RunD and Alpine Linux [3] for others, as the guest operating systems in the microVM for a low memory footprint.

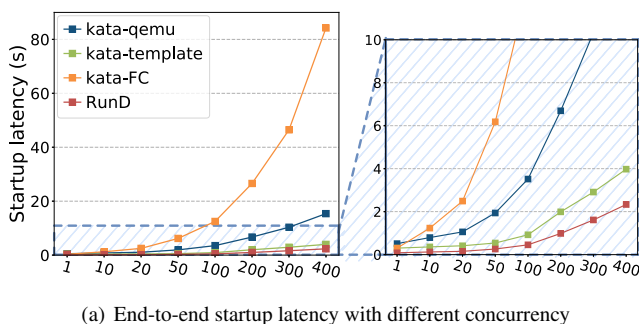
Measurement: in the CRI specification [6], a pod sandbox refers to a microVM with a lightweight pause container [12]. In all the tests, we only create the pod sandboxes without other containers inside, through the `crictl` command. In the following evaluations, the memory specification of a container denotes the size of memory that can be used by itself. The actual memory usage of a container is collected using the `smem` command.

As RunD is proposed to maximize the supported container startup concurrency and deployment density, in the experiment, we start empty secure containers without user codes or data considering that it is a common practice in FaaS to start empty containers concurrently for prewarming. The in-production results show the performance of RunD for actual workloads with all the steps involved.

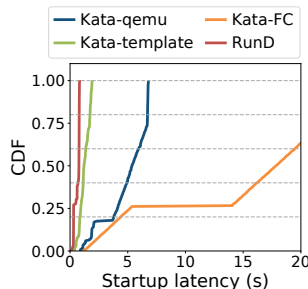
5.2 Concurrent Startup Measurement

In this experiment, we focus on three critical metrics related to user experience: (1) the time needed to start a large number of sandboxes concurrently, (2) the startup latency distribution of the sandboxes, and (3) the CPU overhead on the host. The first metric reveals the throughput of starting sandboxes, and the second metric reveals the experience of every user.

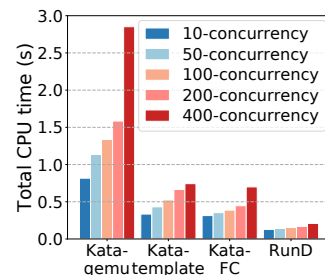
As for the first metric, Figure 10(a) shows the time needed to start a large number of sandboxes concurrently. In the fig-



(a) End-to-end startup latency with different concurrency



(b) Latency distribution



(c) CPU time

Figure 10: The startup metrics with different runtime and concurrency: (a) The end-to-end latency of concurrent startups. The right figure is an enlargement of the left one ($y \in [0, 10]$). (b) The CDF of startup latencies from a 200-way concurrent launch. (c) The CPU usage of concurrent startups.

ure, the x -axis shows the number of sandboxes to be started concurrently, the y -axis shows the overall time needed to startup all the sandboxes.

As shown in the figure, RunD uses the shortest time to start a large number of sandboxes for all concurrency levels. When 200 containers are created concurrently (we already observe such high-concurrency in Alibaba serverless platform), Kata-FC, kata-qemu, kata-template, and RunD needs 47.6s, 6.85s and 2.98s and 1s to create them. Kata-FC requires a much longer time to startup the sandboxes when the concurrency is high. This is because Kata-FC uses *virtio-blk* to create *rootfs*, and the performance is poor at high-concurrency, as we measured in Section 3. There is no such bottleneck in Kata-template and Kata-qemu. Kata-template simply uses template to reduce the overhead of guest kernel and *rootfs* loading, but the inefficient *rootfs* mapping, code self-modification and high host-side overhead of the cgroup operations still exists. As a result, it performs worse than RunD at high startup concurrency. The overall optimizations suggest that RunD provides the performance improvement of about 40% over its nearest baseline, Kata-template, at high-concurrency (e.g., 400-way) startup.

As for the second metric, Figure 10(b) shows the latency distribution of starting each sandbox, when 200 sandboxes are started concurrently. RunD and Kata-template are able to start sandboxes in a stable short time, but the latencies of starting sandboxes with others are out of expected. Users can have identical good experiences with RunD.

As for the CPU overhead, Figure 10(c) shows the CPU time needed on the host to startup sandboxes. When the concurrency is high, RunD greatly reduces the CPU overhead. For instance, when 200 sandboxes are started concurrently, RunD reduces 89.3%, 74.5% and 62.1% CPU overhead compared with Kata-qemu, Kata-template, and Kata-FC, respectively. In addition, the CPU overhead of RunD only increases slightly, when the concurrency increases. This is due to the read/write split policy and the reduction of compute-intensive operations in cgroups. Therefore, RunD

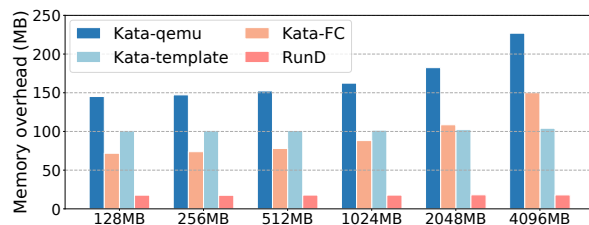


Figure 11: The memory overhead of Kata-qemu, Kata-template, Kata-FC, and RunD (100 sandboxes are deployed).

is scalable in starting more sandboxes concurrently.

In summary, RunD is able to start a single sandbox in 88ms and launch 200 sandboxes simultaneously within 1s, with minor latency fluctuation and CPU overhead.

5.3 Deployment Density

In this experiment, we evaluate the effectiveness of RunD in increasing the sandbox deployment density. In general, the memory used by each container determines the deployment density, while the CPU time needed by each function invocation is minor in the serverless platform. Figure 11 shows the memory overhead when 100 sandboxes are deployed on the experimental node. In the figure, the x -axis shows the memory specification of each sandbox.

As observed, RunD has the least memory overhead among four runtimes, and does not increase with the memory specification. The memory overhead is less than 20MB per sandbox with RunD. Compared to kata-qemu, kata-template and kata-FC, the overhead of RunD is reduced by 54.9%, 27.2%, and 18.9%, respectively, even when the memory specification is 128MB. The memory overhead does not increase, because the microVM template technique uses the on-demand memory loading for the containers. Therefore, the page table required for memory management is determined by the actually used memory space. On the contrary, the memory overheads introduced by Kata-qemu and Kata-FC increase

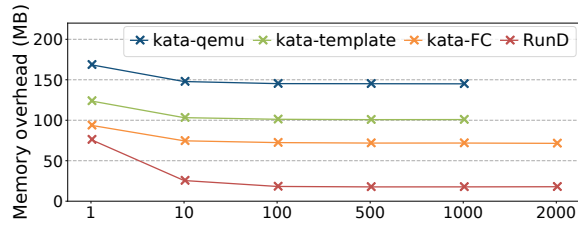


Figure 12: The memory overhead and the amortization by multiple secure containers. The missing point around 2,000 indicates the over-subscription for physical memory space.

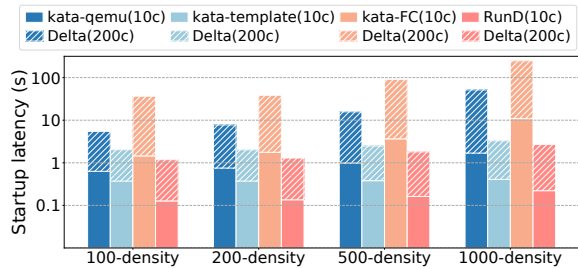


Figure 13: The end-to-end startup latency at different deployment densities. (10c/200c means a 10/200-way concurrent startup, and the Delta means the overhead increment compared with a 10-way concurrent startup).

with larger memory specifications, as the page table is built for all available memory. In addition, the pre-patched kernel image in RunD further reduces memory overhead.

Figure 12 shows the average memory overhead of the sandboxes when different numbers of sandboxes are deployed on a node. The x -axis shows the deployment density. As observed, the average memory overhead reduces with the deployment density, as the sandboxes share the mapped code/data segments. RunD reduces the memory overhead by 87.7%, 82.4%, and 75.1% when 1,000 sandboxes are deployed, respectively, compared with kata-qemu, kata-template, and kata-FC.

RunD supports to deploy over 2,500 sandboxes of 128MB memory specification on the node with 384GB memory.

5.4 Impact of Deployment Density on Startup Latency and Concurrency

When some sandboxes are already deployed on a node, the performance of starting sandboxes concurrently is affected. Figure 13 shows the time needed to boot 10 and 200 sandboxes, when some sandboxes are already deployed on the node. The x -axis shows the number of already deployed sandboxes. The y -axis is in the log10 scale.

When 1,000 sandboxes are already deployed, the time needed to startup 10 containers increases by 1.69s, 0.41s,

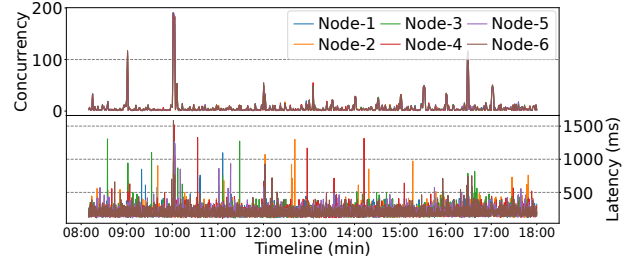


Figure 14: The startup latency and concurrency tracing of RunD in Alibaba serverless platform.

10.8s, and 0.22s compared with the cases in Figure 10(a) with Kata-qemu, Kata-template, Kata-FC, and RunD. In addition, the time needed already increases with the number of already deployed sandboxes.

We can also observe that, the time needed to start 200 sandboxes is at least 10 times as much as that needed to start 10 sandboxes at a 1,000-density deployment in all the tests. The significant increase originates from a large number of cgroups in the host operating system. Scheduling and managing containers with these cgroups consume more CPU cycles, thus resulting in CPU bottlenecks appearing earlier than a low-density deployment. The increased time is the smallest with RunD, because it already eliminates many time-consuming cgroup operations.

RunD shows better performance and stability in supporting high-concurrency startups at high-density deployment.

5.5 In-Production Usage for Serverless

Currently, Alibaba serverless computing platform has adopted RunD. The platform serves almost 4 billion invocations from more than 1 million different functions per day.

Figure 14 reports the sandbox startup concurrency and the corresponding startup latency from six nodes. The specification of each node is the same as our experimental setup in Table 1. The data is collected between 08:00 and 18:00 of Jan 10th, 2022. There are about 800 active sandboxes on each node, when the concurrency data is collected. The in-production startup latency of sandboxes at high-concurrency is consistent with that reported in Section 5.4.

As observed from the figure, the startup concurrency bursts at the beginning of each hour. At most 191 sandboxes are started concurrently around 10:00. RunD starts the 191 sandboxes in 1.6 seconds. We look into the function invocation logs, and find that the periodic burst is caused by the an-hour time trigger and cluster-level load balancing. The periodical burst is pervasive, as the Azure serverless platform traces [14] show the same pattern. In the figure, the sandbox startup latency occasionally increases when the concurrency is low. The long time results from the operation in loading large-scale workloads from the tenants. Although the startup

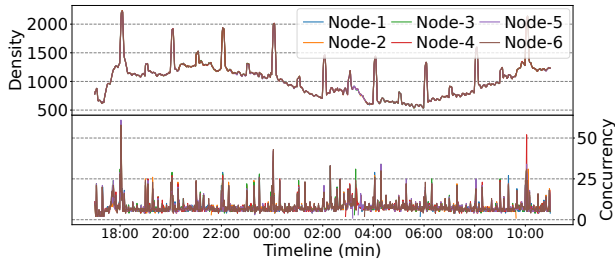


Figure 15: The deployment density and concurrency of RunD in 1 minute intervals, from Alibaba production traces.

concurrency is not always high, it is crucial to ensure a quick startup for a good user experience.

Figure 15 shows the deployment density of the sandboxes on each node. We collect the density statistics of the six nodes between 18:00 of Jan 10th to 10:00 of Jan 11th, 2022. As observed, more than 2,000 sandboxes are deployed on a node at most. We can also find that the high-density deployment happens at the same time as high concurrent startups. This is because many tasks are triggered at the beginning of each hour. The deployment density does not achieve the theoretical upper limit of $384GB/(128+20)MB=2656$ containers, as some functions use more than 128MB memory, and the workloads are also balanced to other nodes.

RunD is production-verified to meet the high-concurrency startup and high-density deployment requirements.

5.6 Lessons Learned from Production Usage

Besides the RunD secure container, we have some insights about designing secure containers for serverless systems.

Lesson-1: the CRI specification designed for Kubernetes is not suitable for serverless system. In CRI, multiple related containers can co-locate in the same sandbox, and a lightweight pause container is started first to prepare the cgroups for the remaining containers. This pause container-based solution is negative for serverless computing, as each sandbox only has a single container for security and privacy.

Lesson-2: Functions tend to use the same standard guest environment provided by the serverless platform. In this case, the language environment (e.g., JVM) can also be integrated into the microVM template. However, the language-level template will invalidate the on-demand memory loading in the guest because some language runtimes need to pre-allocate the available memory. There are tradeoffs between higher memory utilization and less startup time, and the decision should be made based on how often the functions share the language environment.

Lesson-3: The memory usage of user functions is the key aspect determining the upper limit of the deployment density. Most functions are lightweight. When 2,000 sandboxes are deployed on a node of our serverless platform, the CPU

utilization does not achieve 50%, and there is no complaint on the poor performance from users. One reason for the low CPU utilization is that many sandboxes are actually idle and “kept-alive” after its function invocation is completed in a serverless scenario.

6 Related Work

The most closely related work to RunD is FireCracker [20], which proposes a lightweight VMM for serverless runtime. It provides fast startup within 125ms, allowing 150 VMs to start concurrently per second per node, with less than 5MB footprint per VM. However, FireCracker only serves as the hypervisor stack in the Security Container model, without other complex related processes, e.g., *rootfs* [52]. By contrast, RunD investigates the guest-to-host solution through all stacks and provides higher concurrency and density.

Higher-density deployment. Regarding serverless computing, in the space of higher function deployment density of Secure Containers and VMs [57], the key is designing a more lightweight container runtime both in guest and host. Unikernel [36, 37, 43, 47] runs as a built-in GuestOS without necessary add-ons, demonstrating great potential for deploying containers with less overhead. Kuo [33] Explores lightweight guest kernel configurations for use in Unikernel environments, which has similarity to the approach towards reducing guest kernel size. However, Unikernel is hard to be changed once after compilation with the application. Its compile-time invariance results in poor flexibility in practice. SAND [21] adopts the *multi-container-per-VM* model to amortize the memory footprint of sandboxing. However, they do not further investigate the utilization impact of memory fragmentations in a real-system with high-density deployment. Gsight [61] observes that fine-grained function-level profiling can expose more predictability system-level features in the partial interference. With a more accurate interference predicting [27, 44], the function density can get improved with QoS guaranteed.

The above studies make sense in improving the effective density with less interference for serverless. They are orthogonal to our work, because RunD is motivated to improve the maximum deployment density on a single node.

Higher-concurrency startup. In the space of higher function startup concurrency, recent approaches leverage the container prewarm pool [9, 40, 49, 58]. The state-of-the-art on container prewarming, SOCK [42], uses a benefit-to-cost model to select packages pre-installed in zygotes, and builds a tree cache to ensure that the forked zygote container does not import any additional packages other than the private ones the handler specifies. The C/R (Checkpoint/Restore) [7, 31, 39] supporting the VM snapshotting [10, 28, 29, 41, 54] captures the state of a running instance as a checkpoint, and then restores it once cold startup. Observing that most functions only access a small fraction of the files and mem-

ory loaded in the initialization stage, Catalyzer [29] and Replayable Execution [55] extend the C/R mechanism to achieve a faster on-demand recovery and paging when start containers. REAP [52] identifies the guest-side page when loading a VM snapshot and records the metadata during the record phase. Then, for subsequent invocations, REAP proactively prefetches and load the recorded pages into the guest memory for faster and higher-concurrency startup.

The above studies reduce the startup and recovery phases to partially improve the capability of higher-concurrency startup. From a different angle, RunD holistically focuses on prominent bottlenecks through a guest-to-host investigation when start secure containers with high-concurrency. We also proposes a lightweight serverless runtime that production-verified in practice.

7 Conclusion

In serverless computing, the lightweight and short-term functions leads to the requirement of high-density container deployment and high-concurrency container startup. This work dives into the bottlenecks from the entire software stack and proposes RunD, a lightweight secure container runtime for serverless through a holistic guest-to-host solution. The evaluation results and in-production usage prove the efficiency of RunD to launch 200 secure containers in one second, and deploy over 2,500 secure containers per node. RunD is used in Alibaba production serverless platform, and shows good performance in terms of high-density deployment and high-concurrency startup.

Acknowledgment

We would thank Tianlong Wu, Haoran Yang, Xing Di, Xianbin Tang, Tao Ma, Jiang Liu, Zhiyuan Hou, Lei Wang, Zheng Liu, Gang Deng and Huaixin Chang from Alibaba Group for their contributions to this work. We also thank Jon Howell and our anonymous reviewers, for their helpful comments and suggestions.

This work is partially sponsored by the National Natural Science Foundation of China (62022057, 61832006, 61872240), and Shanghai international science and technology collaboration project (21510713600). Quan Chen and Minyi Guo are the corresponding authors.

References

[1] gvisor: Protecting gke and serverless users in the real world. cloud.google.com/blog/products/containers-kubernetes/how-gvisor-protects-google-cloud-services. ..., 2020.

[2] Alibaba function compute. <https://alibabacloud.com/product/function-compute>, 2021.

[3] Alpine linux. <https://www.alpinelinux.org>, 2021.

[4] Aws lambda. <https://aws.amazon.com/lambda/>, 2021.

[5] Aws lambda: The state of serverless. <https://www.datadoghq.com/state-of-serverless-2020/>, 2021.

[6] Container runtime interface (cri) - a plugin interface which enables kubelet to use a wide variety of container runtimes. <https://github.com/kubernetes/cri-api>, 2021.

[7] Criu: A utility to checkpoint/restore linux tasks in userspace. <https://github.com/checkpoint-restore/criu>, 2021.

[8] Device-mapper. <http://www.sourceware.org/dm/>, 2021.

[9] Execute mode in fission. <https://docs.fission.io/docs/usage/executor/>, 2021.

[10] Firecracker snapshotting. <https://github.com/firecracker-microvm/firecracker/blob/master/docs/snapshotting/snapshot-support.md>, 2021.

[11] Google cloud functions. <https://cloud.google.com/functions>, 2021.

[12] Lightweight pause container. https://groups.google.com/g/kubernetes-users/c/jVjv0QK4b_o, 2021.

[13] Microsoft azure functions. <https://azure.microsoft.com/en-us/services/functions>, 2021.

[14] Microsoft azure functions traces. <https://github.com/Azure/AzurePublicDataset>, 2021.

[15] Qemu. <https://www.qemu.org>, 2021.

[16] virtio-fs. <https://virtio-fs.gitlab.io>, 2021.

[17] What is vm templating and how to enable it. <https://github.com/kata-containers/kata-containers/blob/main/docs/how-to/what-is-vm-templating-and-how-do-I-use-it.md>, 2021.

[18] Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022. USENIX Association.

- [19] Kata containers - open source container runtime software. <https://katacontainers.io/>, 2022.
- [20] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 419–434. USENIX Association, 2020.
- [21] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: towards high-performance serverless computing. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 923–935. USENIX Association, 2018.
- [22] S. Barlev, Z. Basil, S. Kohanim, R. Peleg, S. Regev, and Alexandra Shulman-Peleg. Secure yet usable: Protecting servers and linux containers. *IBM J. Res. Dev.*, 60(4):12, 2016.
- [23] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 41–46. USENIX, 2005.
- [24] Guillaume Bonfante, Jean-Yves Marion, and Daniel Reynaud-Plantey. A computability perspective on self-modifying programs. In Dang Van Hung and Padmanabhan Krishnan, editors, *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, pages 231–239. IEEE Computer Society, 2009.
- [25] Marcus Botacin, Marco Antonio Zanata Alves, and André Grégio. The self modifying code (smc)-aware processor (SAP): a security look on architectural impact and support. *J. Comput. Virol. Hacking Tech.*, 16(3):185–196, 2020.
- [26] Rajkumar Buyya, Satish Narayana Srirama, Giuliano Casale, Rodrigo N. Calheiros, Yogesh Simmhan, Blesson Varghese, Erol Gelenbe, Bahman Javadi, Luis Miguel Vaquero, Marco A. S. Netto, Adel Nadjaran Toosi, Maria Alejandra Rodriguez, Ignacio Martín Llorente, Sabrina De Capitani di Vimercati, Pierangela Samarati, Dejan S. Milojicic, Carlos A. Varela, Rami Bahsoon, Marcos Dias de Assunção, Omer Rana, Wanlei Zhou, Hai Jin, Wolfgang Gentsch, Albert Y. Zomaya, and Haiying Shen. A manifesto for future generation cloud computing: Research directions for the next decade. *ACM Comput. Surv.*, 51(5):105:1–105:38, 2019.
- [27] Quan Chen, Shuai Xue, Shang Zhao, Shanpei Chen, Yihao Wu, Yu Xu, Zhuo Song, Tao Ma, Yong Yang, and Minyi Guo. Alita: comprehensive performance isolation through bias resource management for public clouds. In Christine Cuicchi, Irene Qualters, and William T. Kramer, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 32. IEEE/ACM, 2020.
- [28] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In Amin Vahdat and David Wetherall, editors, *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings*. USENIX, 2005.
- [29] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 467–481. ACM, 2020.
- [30] Dawson R. Engler, M. Frans Kaashoek, and James W. O’Toole Jr. Exokernel: An operating system architecture for application-level resource management. In Michael B. Jones, editor, *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, pages 251–266. ACM, 1995.
- [31] Scott Hendrickson, Stephen Sturdevant, Edward Oakes, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. *login Usenix Mag.*, 41(4), 2016.
- [32] Ricardo Koller and Dan Williams. An ounce of prevention is worth a pound of cure: Ahead-of-time preparation for safe high-level container interfaces. In Daniel Peek and Gala Yadgar, editors, *11th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2019, Renton, WA, USA, July 8-9, 2019*. USENIX Association, 2019.

- [33] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A linux in unikernel clothing. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 11:1–11:15. ACM, 2020.
- [34] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, BingSheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Comput. Surv.*, dec 2021. Just Accepted.
- [35] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. Faasflow: enable efficient workflow execution for function-as-a-service. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 782–796. ACM, 2022.
- [36] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: library operating systems for the cloud. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 461–472. ACM, 2013.
- [37] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 218–233. ACM, 2017.
- [38] Massimiliano Mattetti, Alexandra Shulman-Peleg, Yair Allouche, Antonio Corradi, Shlomi Dolev, and Luca Foschini. Securing the infrastructure and the workloads of linux containers. In *2015 IEEE Conference on Communications and Network Security, CNS 2015, Florence, Italy, September 28-30, 2015*, pages 559–567. IEEE, 2015.
- [39] M. Garrett McGrath and Paul R. Brenner. Serverless computing: Design, implementation, and performance. In Aibek Musaev, João Eduardo Ferreira, and Teruo Higashino, editors, *37th IEEE International Conference on Distributed Computing Systems Workshops, ICDCS Workshops 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 405–410. IEEE Computer Society, 2017.
- [40] Anup Mohan, Harshad Sane, Kshitij Doshi, and Saikrishna Edupuganti. Agile cold starts for scalable serverless. In Christina Delimitrou and Dan R. K. Ports, editors, *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, Renton, WA, USA, July 8, 2019*. USENIX Association, 2019.
- [41] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 391–394. USENIX, 2005.
- [42] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: rapid task provisioning with serverless-optimized containers. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 57–70. USENIX Association, 2018.
- [43] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In Jennifer B. Sartor, Mayur Naik, and Chris Rossbach, editors, *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2019, Providence, RI, USA, April 14, 2019*, pages 59–73. ACM, 2019.
- [44] Pu Pang, Quan Chen, Deze Zeng, and Minyi Guo. Adaptive preference-aware co-location for improving resource utilization of power constrained datacenters. *IEEE Trans. Parallel Distributed Syst.*, 32(2):441–456, 2021.
- [45] Rob Pike, David L. Presotto, Sean Dorward, Bob Flandra, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. *Comput. Syst.*, 8(2):221–254, 1995.
- [46] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.
- [47] Florian Schmidt. uniprof: A unikernel stack profiler. In *Posters and Demos Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 31–33. ACM, 2017.
- [48] Mohammad Shahradsad, Jonathan Balkind, and David Wentzlauff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, pages 1063–1075. ACM, 2019.

- [49] Mohammad Shahradd, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Riccardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 205–218. USENIX Association, 2020.
- [50] Vasily Tarasov, Lukas Rupperecht, Dimitris Skourtis, Wenji Li, Raju Rangaswami, and Ming Zhao. Evaluating docker storage performance: from workloads to graph drivers. *Clust. Comput.*, 22(4):1159–1172, 2019.
- [51] Vasily Tarasov, Lukas Rupperecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, NagaPrasad Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. In search of the ideal storage configuration for docker containers. In *2nd IEEE International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2017, Tucson, AZ, USA, September 18-22, 2017*, pages 199–206. IEEE Computer Society, 2017.
- [52] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 559–572. ACM, 2021.
- [53] William Viktorsson, Cristian Klein, and Johan Tordsson. Security-performance trade-offs of kubernetes container runtimes. In *28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2020, Nice, France, November 17-19, 2020*, pages 1–4. IEEE, 2020.
- [54] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In Andrew Herbert and Kenneth P. Birman, editors, *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 148–162. ACM, 2005.
- [55] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 39:1–39:16. ACM, 2019.
- [56] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. Peeking behind the curtains of serverless platforms. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 133–146. USENIX Association, 2018.
- [57] Andrew Whitaker, Marianne Shaw, and Steven Gribble. Denali: Lightweight virtual machines for distributed and networked applications. 03 2002.
- [58] Zhengjun Xu, Haitao Zhang, Xin Geng, Qiong Wu, and Huadong Ma. Adaptive function launching acceleration in serverless computing platforms. In *25th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2019, Tianjin, China, December 4-6, 2019*, pages 9–16. IEEE, 2019.
- [59] Shuai Xue, Shang Zhao, Quan Chen, Gang Deng, Zheng Liu, Jie Zhang, Zhuo Song, Tao Ma, Yong Yang, Yanbo Zhou, Keqiang Niu, Sijie Sun, and Minyi Guo. Spool: Reliable virtualized nvme storage pool in public cloud infrastructure. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 97–110. USENIX Association, 2020.
- [60] Yang Zhan, Alexander Conway, Yizheng Jiao, Nirjhar Mukherjee, Ian Groombridge, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. How to copy files. In Sam H. Noh and Brent Welch, editors, *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 75–89. USENIX Association, 2020.
- [61] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. Understanding, predicting and scheduling serverless workloads under partial interference. In Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin, editors, *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, pages 22:1–22:15. ACM, 2021.

A Artifact Appendix

A.1 Abstract

We choose Kata containers and its three configurations *kata-gemu*, *kata-FC*, *kata-template* as baselines for comparison with RunD. For measuring the startup latency, we use the

crictl command to start pod sandboxes and measure the time between the first *crictl runp* invocation and the last ready pod sandbox. For measuring the memory footprint, we use the *smem* command and the PSS column of its output. All the tests are run on a machine with 104 vCPUs and 384GB of memory running CentOS7.

A.2 Artifact Check-list (Meta-information)

- **Run-time environment:** Alibaba ECS instance;
- **Hardware:** Intel Xeon(Cascade Lake) Platinum 8269CY, CPU and Memory: 104 cores and 384GiB, Storage: Two ESSDs (100GB + 500GB);
- **Software:** Aliyun Cloud OS 2, with Linux kernel 4.19.91, Kata container 1.12.1 and 2.2.3, containerd 1.3.10, smem 1.4;
- **Metrics:** average latency and average memory footprint;
- **Time is needed to complete experiments:** 10 hours;
- **Available:** https://github.com/chengjiagan/RunD_ATC22
- **Code Licenses:** Apache-2.0 license

A.3 How to Access and Installation

Github Link: https://github.com/chengjiagan/RunD_ATC22. Then you should follow the *README* instructions to get installation.

A.4 Experiment Workflow

A.4.1 High-concurrency Experiment (Section 5.2)

Scripts are provided to run the high-concurrency test for kata-qemu, kata-fc and kata-template. To run high-concurrency tests:

```
$ ./script/time_kata_test.sh
$ ./script/time_katafc_test.sh
$ ./script/time_katemplate_test.sh
```

They may take several hours to finish. Some concurrency tests can be removed by removing the corresponding concurrency setting in file *time_test.conf* to shorten the time. The scripts will create a directory (e.g., named like *time_kata_05120948*) to store the logs.

We provide python scripts to analyze logs from the tests:

```
$ python3 data/time.py
$ python3 data/cpu.py
```

The python script will create two .csv files in the result directory: *time.csv* and *cpu.csv*. Each line in the csv file indicates the average cold-start latency and cpu time of a container runtime.

A.4.2 High-density Experiment (Section 5.3)

Scripts are provided to run the high-density test for kata-qemu, kata-fc and kata-template. To run high-density tests:

```
$ ./script/mem_kata_test.sh
$ ./script/mem_katafc_test.sh
$ ./script/mem_katemplate_test.sh
```

Density and memory capacity of containers in the tests can be changed in the file *mem_test.conf*. The scripts will create a directory named like *mem_kata_05120948* to store the logs.

We provide a python script to analyze logs from the tests:

```
$ python3 data/mem.py
```

The python script will create a csv file for each runtime, named like *mem_kata.csv*, containing the average memory consumption of containers with different memory capacity in different density.

A.4.3 Density Impact on Concurrency (Section 5.4)

Scripts are provided to run the high-density test for kata-qemu, kata-fc and kata-template. To run tests:

```
$ ./script/density_kata_test.sh
$ ./script/density_katafc_test.sh
$ ./script/density_katemplate_test.sh
```

The background density and the concurrency of the tests can be changed in the file *density_test.conf*. The scripts will create a directory (e.g., named like *density_kata_05120948*) to store the logs.

We provide a python script to analyze logs from the tests:

```
$ python3 data/density.py
```

The python script will create a csv file for each runtime, named like *density_kata.csv*, containing the average cold-start latency under different background densities and concurrencies.

A.5 Expected Results and Notes

The expect results are all stored in *ae_data* directory. Considering that some related binary packages are tightly integrated with our internal system, we provide a screencast *ATC_RunD_AE.mp4* of the tool along with the results. You can also find RunD-related performance and execution logs in our artifact..

Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing

^{1,3}Zijun Li, ¹Linsong Guo, ¹Quan Chen, ¹Jiagan Cheng, ¹Chuhao Xu, ²Deze Zeng, ³Zhuo Song, ³Tao Ma, ³Yong Yang, ¹Chao Li, ¹Minyi Guo

¹Department of Computer Science and Engineering, Shanghai Jiao Tong University

²School of Computer Science, China University of Geosciences

³Alibaba Cloud

Abstract

In serverless computing, each function invocation is executed in a container (or a Virtual Machine), and container cold startup results in long response latency. We observe that some functions suffer from cold container startup, while the warm containers of other functions are idle. Based on the observation, other than booting a new container for a function from scratch, we propose to alleviate the cold startup by re-purposing a warm but idle container from another function. We implement a container management scheme, named **Pagurus**, to achieve the purpose. Pagurus comprises an intra-function manager for replacing an idle warm container to be a container that other functions can use without introducing additional security issues, an inter-function scheduler for scheduling containers between functions, and a sharing-aware function balancer at the cluster-level for balancing the workload across different nodes. Experiments using Azure serverless traces show that Pagurus alleviates 84.6% of the cold startup, and the cold startup latency is reduced from hundreds of milliseconds to 16 milliseconds if alleviated.

1 Introduction

Owing advantages of high maintainability and testability, serverless computing is suitable for the ever-growing Internet services (the tenants are charged per invocation). As a result, hyperscalers now provide serverless computing services (e.g., Amazon Lambda [5], Google Cloud Function [11], Microsoft Azure Functions [12], and Alibaba Function Compute [1]). Meanwhile, some open-source serverless computing solutions like Apache OpenWhisk [3] and Fission [9] have also been developed and released.

In serverless computing, user functions run in containers (or Virtual Machines), and the containers are specialized for a function (a container is not allowed to run different user functions). Warm containers refer to the keep-alive containers that serve subsequent invocations (*warm startup*). If there is no warm container for a function invocation, a new container is started from scratch (*cold startup*). The cold startup

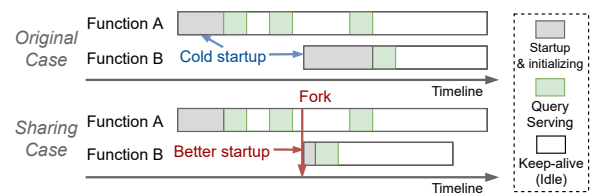


Figure 1: “Fork” an idle container of Function A for Function B to alleviate its cold startups.

latency is more than $10\times$ of the warm startup latency due to the container creation, software environment setup, and code initialization [40, 44, 47, 49, 59, 61]. It is ideal if all functions can run in warm containers. However, keep-alive containers are recycled to save resources once no new invocation arrives during the lifetime.

Many efforts have been devoted to speeding up the container cold startup [7, 28, 31, 32, 45, 46, 54–56, 58]. The prewarm-based methods create containers and runtime in advance, one method of which is prewarming customized containers for each function that includes all its required software packages [8, 15, 27, 28, 60]. However, it brings heavy memory consumption. Another method of prewarming containers is only installing common packages, and all functions share the prewarmed container pool [14, 45, 46]. This method is more memory space-efficient, but generating customized containers for a function from the prewarmed containers suffers from package download and installing latency overhead. Current solutions mainly adopt the second method [14, 46].

To alleviate the memory waste and minimize the function response time, instead of prewarming containers, we propose to alleviate the cold container startup through container sharing. For instance, if function A in Figure 1 can “fork or lend” the runtime checkpoint from its idle warm containers for function B, the cold startup of B is eliminated. By such means, we can leverage a function’s idle warm containers before being recycled to help others that tend to experience cold container startup.

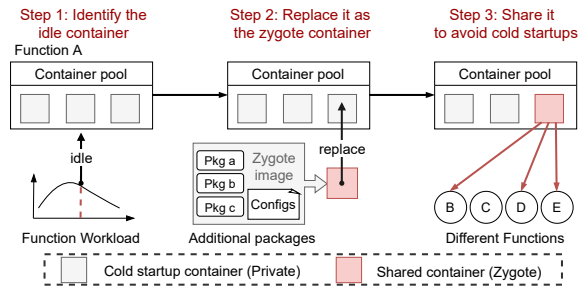


Figure 2: The container sharing logic and challenging steps.

Analyzing the container sharing procedure, we find three challenges in achieving such goal. **1) The function invocation loads are not stable [51].** It is difficult to identify whether function A’s warm containers are actually idle or not. If the warm containers of A are always used to help B, A’s invocation may not be able to get a warm container and may even suffer from Quality-of-Service (QoS) violation. **2) Functions rely on different software packages.** When function A “forks” a warm container for function B, the warm container has to install extra packages. Installing and importing these packages may take longer than the cold container startup. Worse still, greedy re-packaging for excessive functions can also lead to huge image size or incur package version contradictions. **3) “Forking” an idle container from other functions may introduce security vulnerabilities.** While a function’s code or data are stored privately, sharing the container with other functions is risky.

We propose a container management system, named **Pagurus** to tackle these challenges. Figure 2 shows the steps of using function A’s idle warm containers to alleviate the cold container startup of other functions. The key idea is to replace its idle containers with new containers that other functions can safely use. The new container is created through a new image that already installs the required software packages of other functions, without including the code/data of the original function A. In this way, the warm containers of a function are classified into three categories: *private containers*, *zygote containers*, and *helper containers*. A function’s private containers can only be used by itself. Its zygote containers are the new containers that other functions can use. Its helper containers are the containers forked from other functions’ zygotes. A helper container already loads its user code for future invocations. By using privilege control in the operating system, a function that uses a helper container cannot obtain any code, data, or package information of other functions.

Pagurus uses an *intra-function container manager* for each function to manage its containers, an *inter-function scheduler* on each node to manage the “fork” action between functions, and a *sharing-aware function balancer* to schedule functions across the nodes. For a function, the

intra-function manager monitors the status of each container, identifies idle warm containers, and re-purposes an idle container based on a QoS-based timer. The inter-function scheduler, acting as an orchestrator, determines the to-be-helped functions of each function. We design a Similarity Filtered Weighted Random Sampling (SF-WRS) algorithm to find an appropriate set of to-be-helped functions. Besides, the sharing-aware function balancer distributes function invocations to different nodes to achieve efficient inter-function container sharing.

Pagurus requires no offline analysis or profile on the functions, thus can be easily adopted in production. The main contributions of this paper are as follows.

- **A resource-friendly design of zygote and helper container.** Zygote container enables resource-saving through package and function reclamation, without incurring any additional security issues meanwhile.
- **The design of a SF-WRS re-packing policy.** Based on the package similarity between functions and the frequency of function cold startups, SF-WRS policy reduces the number of packages to-be-installed, thus minimizing the memory needed and the overhead of creating zygote containers.
- **The design of an efficient container sharing mechanism.** Pagurus divides the warm containers of a function into three types and manages the three types of containers in different ways. The mechanism efficiently alleviates the cold container startup.

We evaluate Pagurus using both best-practice AWS serverless functions [6] and Azure traces [50]. Experiments show that Pagurus alleviates 84.6% of cold startups on average in Azure traces, and the cold startup latency is reduced from hundreds of milliseconds to 16 milliseconds if alleviated.

2 Background and Related Work

If a function is invoked for the first time or there is no alive (or warm) container for it, the serverless system starts a new container to encapsulate its function runtime, initializes the software environment, loads application-specific code, and runs the function. All these steps make up a *cold startup* and may even take several seconds [21, 36, 59]. The cold startup significantly increases queries’ end-to-end latency [23, 33, 47, 49]. The long latency problem worsens when the function invocation is short (e.g., hundreds of milliseconds).

Prewarm startup spawns template containers that are already initialized with the software environment. Though it skips the container startup and users only need to perform application-specific code initialization [2, 3, 32, 46], its pre-loaded packages can either make the image size too large [20, 32, 53], or cause more memory consumption for the prewarm container [24, 46, 50].

Many prior studies have been conducted to reduce the container startup latency [19, 28, 34, 48, 52, 54]. However, existing works mainly focused on seeking lightweight virtualization technologies to pursue lower overhead [17] or optimizing prewarm strategies for more accurate prediction models and less initialization cost []. A common optimization is to pause the container when idle to save resources consumed by function codes and packages, and then reload it for reusing when invoked [34, 44, 45, 57].

SAND [19] separated applications via containers while allowing functions of one application to run in the same container by different processes. FaasCache [31] took the caching model for objects into serverless context, and implemented the Greedy-Dual keep-alive caching mechanism to reduce the resource requirement and keep containers warm. Shahrad *et al.* [50] proposed to dynamically change the instance lifetime of the recycling and provisioning instances according to the time series prediction. Some researchers use C/R (Checkpoint and Restore) [7, 55, 56, 58] that restores container images from checkpoints to speed up the cold startup. For example, Catalyzer [28] utilized C/R to realize on-demand recovery. However, it still incurs long latency compared with a warm startup. The above technologies are orthogonal to us, and Pagurus can be combined with them to reduce the cold startup latency further. SOCK [46] introduced a tree cache and uses the benefit-to-cost model to update packages in the prewarmed containers dynamically, but the zygote design consumes more memory when maintaining packages by a cache-tree. Moreover, the cache-tree does not work if functions require conflicted package versions.

Pagurus resolves the problems through inter-function container sharing with conflict concerns, and needs neither pool-size tradeoffs nor time-consuming model training.

3 Investigation and Motivations

In this section, we discuss the current prewarm-based mechanism for alleviating the cold container startups, and show the possibility of eliminating the cold container startup with inter-function container sharing.

3.1 Latencies of Cold and Prewarm Startups

A cold container startup is done in three time-consuming steps: *create container from the image*, *initialize software environment*, and *initialize application-specific code*. With the prewarm mechanism (used in OpenWhisk [3] and production platforms), several containers that already import common libs/packages are hatched in a container pool. A function invocation with no warm container can specialize the prewarmed container by installing the extra packages.

We use prewarm-enabled OpenWhisk with local cache as the serverless platform, and use the best practice serverless

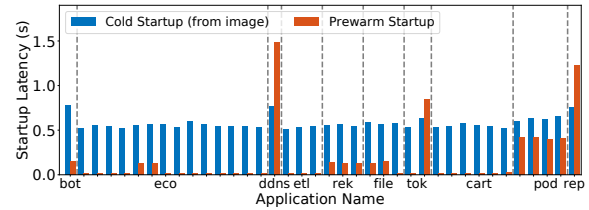


Figure 3: The cold startup latency and prewarm startup latency of the benchmarks in OpenWhisk.

applications in AWS [6] as the benchmarks, to investigate the impacts of cold and prewarm startup on the end-to-end latency of a function invocation. A benchmark may have several functions [41]. As for the hardware, we use one node to perform the computation and one node to generate function invocations. The benchmarks, software, and hardware setups are described in Section 8.

Figure 3 shows the time of generating a container when it is started from the image, or is specialized from a common prewarmed container. As shown, the cold container startup takes about 500 milliseconds. The prewarm startup takes 15 milliseconds in the best case, but takes more than 1500 milliseconds in the worst case (e.g., function *union* in the benchmark *ddns*). This is because *union* requires to load/install many additional packages in the prewarmed containers, and the package loading is time-consuming.

Intuitively, a prewarmed container may install all the software packages required by all the functions on a physical node to speed up the prewarm startup. It is possible because for most serverless systems, the packages needed by a function (besides the private ones) are usually given by its user in a *requirements.txt*, and are publicly accessible for FaaS providers. However, many functions require software packages of contradicting versions. In addition, such a solution may expose the package information of other functions. The pre-imported requirements will implicitly embody user privacy. *Due to the software conflict and privacy concerns, it is not a good option to install packages for all functions in the prewarmed containers.*

3.2 Limitations of Prewarm Schemes

We then explore the effectiveness of the prewarm schemes in alleviating the cold startup. In this experiment, we run all the benchmarks on a single node, and the invocation patterns of the functions are the same as the patterns in the Azure serverless traces. The invocation patterns actually follow the Pareto distribution (most of the invocations are for a small part of the functions) [15]. By default, a prewarm container pool has two prewarmed containers on a node [2].

Figure 4 shows the percentage of the remaining cold startups with the prewarm mechanism. Many cold startups are not eliminated (e.g., functions in *eco* and *cart*). This phe-

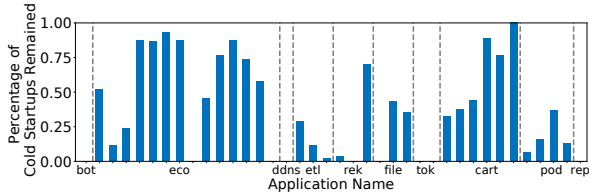


Figure 4: The remained cold startup in prewarm-enabled OpenWhisk compared with the disabled one.

nomenon attributes to the inappropriate pool size of the prewarm container pool. For *eco* and *cart*, their functions are invoked simultaneously/or in short intervals to satisfy complex business logic, such as workflows [18, 22, 25]. For instance, five functions are triggered simultaneously by a caller in *eco*. These functions contend for the prewarmed containers.

It is nontrivial to appropriately configure the prewarm scheme due to two considerations. **1) Pool-size and memory overhead trade-off.** If we prewarm more containers, larger additional memory space is used. In our experiment, the prewarmed containers use more than 1GB of memory (on a node with 16GB memory) to eliminate 80% of the cold startup. The prewarm mechanism is not able to effectively eliminate the cold startup with reasonable memory overhead.

2) User experience and system efficiency contradiction. As discussed, most of the invocations are from a small part of the functions [15], and a prewarmed container can only cache a small number of packages for the low memory overhead. Caching packages for frequently invoked functions improves the system efficiency (frequent invocations have low startup time), but results in poor user experience (invocations of most functions tend to suffer from the long package installation time), and vice versa.

The current container prewarm scheme is not efficient due to several inevitable trade-offs. It is beneficial to alleviate cold startups without trapping in the same dilemmas.

3.3 Opportunity of Reusing Idle Containers

We therefore propose to alleviate cold startup without relying on prewarming containers. The key idea is leveraging the warm but idle containers of some functions to alleviate the cold startups. A function invocation that requires cold startup may “steal” an idle warm container from other functions.

The proposed scheme is effective only when there are idle warm containers in some functions when an invocation tends to suffer from the cold container startup. In principle, only underutilized warm containers that are active due to the keep-alive strategy can be used by other functions. Otherwise, always stealing a warm container directly may result in the cold container startup of the victim function.

We analyze the *day07* trace of Azure serverless platform [50] (the trace contains invocations of over 44,000

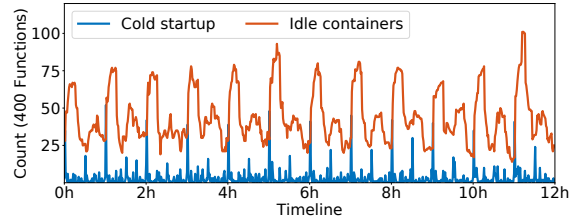


Figure 5: The number of container cold startups and idle containers from 400 randomly selected functions.

functions) to verify the above requirement. If a container triggers recycling, it must be an idle warm container because of no new invocation during its lifetime. By replaying the trace, we find that the warm containers for some functions are idle (no invocation is received during this idle time), while some other functions suffer from cold startup. We refer to idle warm containers as idle containers.

Figure 5 shows the number of idle containers and cold startups when replaying the trace. As observed, the time that idle containers and cold startup happen are similar, and there are more idle containers than the cold startups. If the time does not match, the functions that suffer from cold startup cannot find idle containers from other functions.

The time matches because many containers are prepared and invoked to serve the high load, and they become inactive when the load drops. We can observe a significant discontinuity at the beginning of each hour as there is a certain number of functions with a 1-hour timer trigger, and they are invoked once and will keep idle during the rest of their lifetime. In this case, excessive idle containers are pervasive.

In summary, serverless computing systems usually adopt a keep-alive strategy (e.g., 15 minutes) to reduce the cold startup. The kept-alive containers are idle before they are recycled. The widely-existed diurnal load pattern also makes containers over-provisioned at the high load. These containers will become idle when the load drops as well.

Based on the investigation, we observe the opportunity to leverage the idle containers of some functions to help others that suffer from cold startup on the same node.

4 Design of Pagurus

There are two prerequisites to alleviate the cold container startup with the warm containers of other functions. First, the container manager has to identify the actual idle warm containers. Otherwise, the “steal” results in the container cold startup of the victim function. Second, the proposed strategy should not expose any information of a function (e.g., data, code, package requirements) to other functions from the consideration of security.

We propose and implement **Pagurus**, a container management system that fulfills the two prerequisites. Figure 6

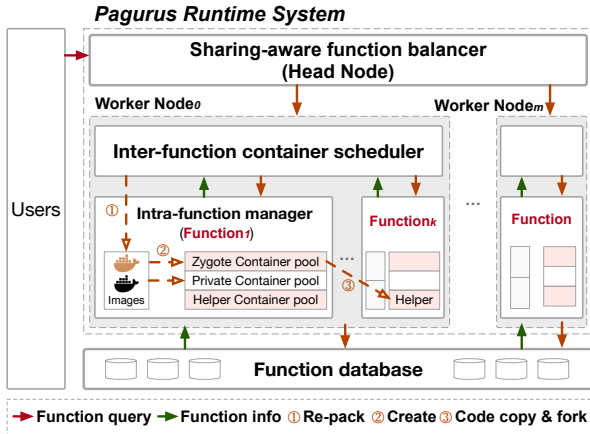


Figure 6: Design of Pagurus.

shows the design of Pagurus. It comprises an *intra-function container manager* for each function, an *inter-function container scheduler* on each node, and a *sharing-aware function balancer* at the cluster level. The intra-function container manager of a function manages its three types of containers (private containers, zygote containers, and helper containers) (Section 5). The inter-function scheduler manages the zygotes sharing between functions (Section 6). The sharing-aware function balancer maps the functions across multiple nodes to minimize the system-wide cold startup (Section 7).

Based on runtime statistics, a function’s idle warm containers are replaced with its zygote containers that are newly created from its zygote image (② in Figure 6). The zygote image does not include the code or data of any functions. Pagurus uses the inter-function container scheduler on each node to generate the zygote image of each function (① in Figure 6). Creating zygote images does not introduce extra runtime latency overhead, as it is done asynchronously before replacing the idle container with a zygote container. The inter-function container scheduler determines the possible to-be-helped functions of each function based on Similarity-Filtered Weighted Random Sampling (SF-WRS) policy, which will be detailed in Section 6.1. A function’s zygote container additionally installs the required packages of its to-be-helped functions in an anonymous fashion. Based on the privilege control of the Linux operating system, a function is only able to access its own packages.

Specifically, when an invocation of a function f arrives, it obtains a container to host the invocation in four steps.

1) It first tries to obtain an idle warm private container from its own private container pool directly. Then, if its private container pool is empty, it checks whether its helper container pool has containers for queries.

2) If both the private pool and the helper pool are empty, it will further check whether its zygote container pool has some containers already adapted for other functions. If not

empty, a zygote container can be used to host the invocation.

3) If its zygote container pool is also empty, Pagurus tries to find a container that includes the required packages of f from other functions’ zygote container pool. The forked zygote then joins the helper container pool of f (③ in Figure 6).

4) If all the above steps fail, the invocation of f would suffer from a cold container startup.

5 Intra-function Container Management

The key points of the intra-function manager are identifying the actual idle containers, and designing an efficient sharing mechanism.

5.1 Identifying Idle Containers

In principle, a container is idle when it does not host function invocations for a long time. For a function f , we introduce a timer in each of its containers to measure the free time. A warm container is treated to be idle if its timer exceeds threshold $T_{idle}(f)$. The timer is reset once the container receives an invocation.

The design principle here is that most function invocations can still get warm containers, even when a container is identified to be idle and “stolen” by other functions. Different functions should have different idle thresholds because of their diverse invocation patterns. We explore the runtime invocation arrival pattern to determine the value of $T_{idle}(f)$ for a function f . Specifically, we use all the m invocations during the container lifetime, and let T_1, T_2, \dots, T_m represent the time intervals between the adjacent invocations (the time interval is sorted in the ascending order). Equation (1) calculates the idle threshold $T_{idle}(f)$ for the function f in the next time period.

$$T_{idle}(f) = \begin{cases} T_{[0.95m]}, & m \geq 30, \\ T_{default}, & m < 30. \end{cases} \quad (1)$$

In the equation, $T_{[0.95m]}$ is the 95%-ile time interval of the m samples. In this case, for frequent invoked functions ($m \geq 30$), more than 95% invocations of f tend to get warm containers, if the invocation arrival patterns remain. We use 30 to be the sampling target for stability considerations. For occasionally invoked functions, $T_{idle}(f)$ is set to be $T_{default}$, and almost all the invocations get warm containers. $T_{default}$ may impact the overall efficiency for idle identification, and Section 8.4.3 evaluates the sensitivity of Pagurus to it.

5.2 Replacing Idle Containers with Zygotes

A function’s idle containers cannot be used by other functions directly, as the data and code of the function may still reside in the memory of the idle containers. To this end, Pagurus creates a zygote container that does not include any

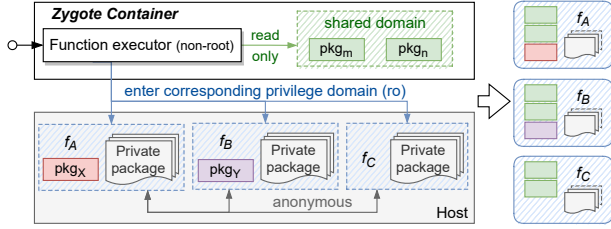


Figure 7: Security assurance of a zygote container.

data or code of the owner function, and uses the zygote container to replace the original idle warm container. The zygote container is created from an image that installs the shared packages of all the to-be-helped functions. Section 6 discusses the policies used to determine the to-be-helped functions and generate the zygote image of a function.

One may be concerned about the security and privacy of the zygote container for re-purposing. Figure 7 shows the way to avoid package information leakage in zygote containers. All the functions run as non-root users [4, 10, 16]. In the figure, f_B and f_C are the to-be-helped functions in f_A 's zygote container. In general, the common intersectant packages required by all functions are installed as a shared domain (pkg_m, pkg_n) in the zygote container, and the additional complementary and private packages of to-be-helped functions are cached in different directories of the host. These directories are mounted anonymously into the zygote, thus ensuring that others cannot identify a function.

Each function that may use the zygote container is given a privilege domain and is only allowed to access its corresponding package directory. The privilege domain and privilege control are provided by Linux operating system and different non-root users. For instance, in Figure 7, when function f_B obtains the zygote container, it can only enter its own privilege domain for f_B 's packages (private packages and pkg_y) to specialize its software runtime. In this way, the zygote container serves as a safety checkpoint. Because it does not import any user-related code and data, the function privacy of the software environment is also protected.

6 Inter-function Container Scheduling

The inter-function container scheduler selects to-be-helped functions for zygotes, re-packs zygote images, and manages the fork operation for helper containers.

6.1 Selecting To-be-helped Functions

A straightforward approach is to treat all the other co-located functions as the candidate to-be-helped functions, and install all the required packages into a zygote image. However, this approach suffers from extremely high re-packing overhead, in terms of both time and resource consumption. When

re-packing a zygote image, we have two important observations. On the one hand, if the set of to-be-installed packages is large, it is time-consuming and resource-unfriendly to create a giant zygote image. On the other hand, some functions tend to have more cold startups than others (as observed from Azure traces [12]), inappropriate selection of to-be-helped functions is inefficient in alleviating the system-wide cold startups. Taking the above challenges into consideration, we propose **SF-WRS** (Similarity Filtered Weighted Random Sampling) algorithm, which contains:

- A Similarity-based Filter to find out to-be-helped candidates based on the similarity of functions' packages. In this way, a zygote installs fewer complementary packages, thereby lowering the re-packing overhead.
- A WRS (Weighted Random Sampling) strategy [29] to select K to-be-helped functions based on the cold startup frequency of each function on the node. Pagurus tends to prepare zygote images for the functions that suffer from more cold startups with high possibility.

Similarity-based Filter. Focusing on the package information, a function f can be viewed as a set of packages, i.e., $f = \{pkg_1, pkg_2, \dots\}$, where pkg_i is assigned in the *requirements.txt* and refers to the required package in f 's runtime environment. Let $\mathbb{F}_n = \{f'_1, f'_2, \dots\}, \forall f'_i \neq f$ represent the set of functions on node n when function f triggers re-packing. The package difference between f and $f'_i \in \mathbb{F}_n$ imposes a deep influence on the re-packing overhead. Let us denote the containment relationship of a package pkg in f_A and another function f_B as

$$con(pkg, f_A) = \begin{cases} 1, & \text{if } pkg \in f_A, \forall pkg \in f_A \cup f_B. \\ 0, & \text{others,} \end{cases} \quad (2)$$

We can then derive the containment relationship vector of f and $f'_i \in \mathbb{F}_n$ as $\vec{f} = \{con(pkg, f) | \forall pkg \in f \cup f'_i\}$ and $\vec{f}'_i = \{con(pkg, f'_i) | \forall pkg \in f \cup f'_i\}$, respectively. The similarity between f and f'_i thus can be calculated as their cosine distance by

$$Cos(\vec{f}, \vec{f}'_i) = \begin{cases} \frac{\vec{f} \cdot \vec{f}'_i}{\|\vec{f}\| \|\vec{f}'_i\|}, & \|\vec{f}\| \|\vec{f}'_i\| \neq 0, \forall f'_i \in \mathbb{F}_n. \\ 1, & \|\vec{f}\| \|\vec{f}'_i\| = 0. \end{cases} \quad (3)$$

Thereafter, we can obtain an initial f 's to-be-helped function candidate set \mathbb{C}_n^f by removing those functions with similarity lower than *TargetSimilarity* from \mathbb{F}_n . *TargetSimilarity* can be set as the median similarity in default.

Note that, as discussed before, a package may be specified in different versions, and a zygote image with version conflict (i.e., the same package but different versions) could not be re-purposed by another function. Denoting the version of a package pkg in function f as $V(pkg, f)$, we can express the version conflict relationship between f and f' as

$$Conflict(f, f') = \begin{cases} 1, & \exists pkg \in f, V(pkg, f) \neq V(pkg, f'), \\ 0, & \text{others.} \end{cases} \quad (4)$$

Then, we first ensure that there is no conflict between function f and its candidates, by removing the functions with version conflict from the candidate set. However, it is still possible that some candidates conflict with each other. For instance, candidates f'_1 and f'_2 do not conflict with f , but f'_1 conflicts with f'_2 . In this case, they cannot be packed into a single zygote image either. The candidate set \mathbb{C}_n^f should be further updated by Equation (5), where $\mathbb{C}_n^f \setminus f'_i$ represents the complement of f'_i in \mathbb{C}_n^f .

$$\begin{aligned} \mathbb{C}_n^f &= \{f'_i | \text{Conflict}(f, f'_i) = 0, \\ &\text{Conflict}(f'_i, \mathbb{C}_n^f \setminus f'_i) = 0, \forall f'_i \in \mathbb{C}_n^f, \} \end{aligned} \quad (5)$$

Take the package conflict in AWS application benchmarks as an example, function `tcp_check_transcribe` requires the package `aws_requests_auth` with version 0.4.1, while function `ep_delivery_on_package_created` requires that with version 0.4.3. It denotes that applications have various package similarities, and two functions may rely on conflicted packages. By selecting the to-be-helped functions based on package similarity, a zygote image installs fewer packages for zygote images with less re-packing overhead.

WRS selection. After the similarity filter and conflict recognition, the to-be-helped function candidates can be significantly reduced. However, it is still nontrivial for the inter-function scheduler to determine the appropriate number of to-be-helped functions without resulting in too large image size, too long image generation time, or failing to eliminate most cold startups. Therefore, we should choose an appropriate number, say K , of functions from candidates \mathbb{C}_n^f that tends to eliminate the cold startups with high probability.

Therefore, we first remove the functions never re-invoked from \mathbb{C}_n^f . Let I be the number of remaining functions that have been invoked more than once. We can calculate K as

$$K = \frac{\sum_{n=1}^I K_n}{I} = \sum_{n=1}^I \left[\frac{\sum_{n=1}^I \text{Cold}(f'_n) + \sum_{n=1}^I \text{Zygote}(f'_n)}{(\text{Cold}(f'_n) + \text{Zygote}(f'_n)) \text{Num}(\text{Zygote})} \right] / I, \quad (6)$$

K_i is the expected number of to-be-helped functions for f'_i , $\text{Num}(\text{Zygote})$ is the average number of active zygotes in the system, $\text{Zygote}(f'_i)$ and $\text{Cold}(f'_i)$ indicate the times a function experiences zygote-based invocation and cold startups of f'_i , respectively. K_i ensures that each to-be-helped function f'_i can be re-packed into a zygote container at least once.

Algorithm 1 summarizes the SF-WRS algorithm. First, the inter-function scheduler filters out the candidate functions with low similarity values (lines 1-3), recognizes the package conflicts (lines 4-6), and then selects K to-be-helped ones by the A-ExpJ algorithm, which is a variation of WRS (Weighted Random Sampling) algorithm [29] (lines 9-12). Compared with naive WRS, A-ExpJ shows much lower time complexity. The time complexity of selecting K functions is $O(K \log(\frac{n}{K}))$ with A-ExpJ, and the time complexity of naive WRS is $O(n)$.

Algorithm 1 SF-WRS Selection Algorithm

Require: To-be-helped function candidates \mathbb{C}_n^f
Require: $\text{Cold}(f'_i)$ and $\text{Zygote}(f'_i)$ of function f'_i in last hour

- 1: $\mathbb{C}_n^f = \text{Sample.init}(\mathbb{F}_n)$
- 2: **for** f'_i **in** \mathbb{C}_n^f **do**
- 3: **if** $\text{Cos}(\vec{f}, f'_i) < \text{TargetSimilarity}$ **then:** $\mathbb{C}_n^f.\text{delete}(f'_i)$
- 4: **for** f'_j **in** \mathbb{C}_n^f **do**
- 5: **if** $\text{Conflict}(f, f'_j) = 1$ **or** $\text{Conflict}(f'_j, \mathbb{C}_n^f \setminus f'_j) = 1$ **then**
- 6: $\mathbb{C}_n^f.\text{delete}(f'_j)$
- 7: **if** $\mathbb{C}_n^f \neq \text{Null}$ **then**
- 8: $\text{Total} = \sum_{n=1}^K \text{Cold}(f'_i) + \sum_{n=1}^K \text{Zygote}(f'_i)$
- 9: **for** f'_k **in** \mathbb{C}_n^f **do**
- 10: $\text{Repack}(f'_k) = [\text{Cold}(f'_k) + \text{Zygote}(f'_k)] / \text{Total}$
- 11: $\text{Sample.append}((f'_k, \text{Repack}(f'_k)))$
- 12: $A\text{-ExpJ}(\text{Sample}, K)$

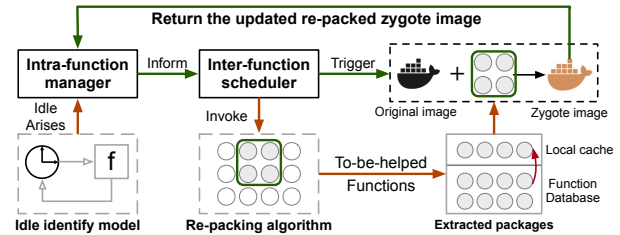


Figure 8: The key steps (represented in green) of re-packing a zygote image for the first time.

6.2 Re-packing a Zygote Image

Figure 8 shows the steps of re-packing the zygote image for a function f . When the intra-function container manager of f identifies an idle container, it informs the inter-function scheduler, then selects the to-be-helped functions of f based on the SF-WRS algorithm. After that, the inter-function scheduler triggers the re-packing operation, obtains the packages, and re-packs the zygote image. Only the packages shared by all the to-be-helped functions are installed in the shared domain. Finally, the re-packed zygote image is returned to the intra-function container manager of f for building zygote containers to replace f 's idle containers.

The inter-function container manager has an advantage compared with the traditional building method, where the image is built through the network from the container repository. Pagurus omits the downloading of the required packages in a zygote image through the network again, benefiting from the locally cached packages, when creating the private container images [26, 39]. By reusing cached packages, re-packing a zygote image takes a much shorter time.

Besides, the zygote image of a function is asynchronously re-packed before its to-be-helped functions actually meet in cold startups. Re-packing a zygote image does not result in long response latencies of to-be-helped function invocations.

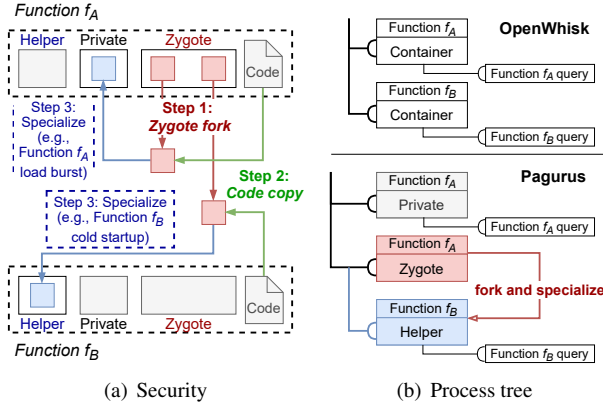


Figure 9: Security guarantee when forking a zygote.

6.3 Forking a Zygote Container

The rest undertaking is to safely and efficiently share zygote containers for other functions. To provide higher availability for multiple to-be-helped functions getting zygote containers, we fork the zygote container to be helper containers, rather than directly specializing it. The zygote container remains there for other to-be-helped functions. The forked containers also ensure software security by the anonymous mounting and privilege domain as zygote containers do.

When a zygote container is forked from function f_A to be f_B 's helper container, the code of f_B is copied into the forked one. We implement two plugins *Zygote fork* and *Code copy*, in the inter-function scheduler. The *Zygote fork* plugin forks a container asynchronously, un-mounts the package directories of functions other than f_B , and transfers the control access to f_B 's corresponding privilege domain. The *Code copy* plugin copies the code of f_B to the helper container.

Figure 9(a) shows the steps of f_B forking a zygote container of function f_A . In Step 1, a zygote container of f_A is forked through the *Zygote fork* plugin. Then, the *Code copy* plugin copies the code of f_B into the forked zygote (Step 2). Lastly, the forked container joins the helper container pool of f_B (Step 3). It also provides process-level isolation for queries, as shown in Figure 9(b).

7 Sharing-aware Function Balancing

In existing serverless computing clusters, hash-based methods or resource usage-based methods are often used to route user queries [13, 30, 35, 37, 38, 42, 43]. It is possible that the functions on a node do not share many packages. In this case, the host node needs to create many private directories with many packages, resulting in poor resource efficiency. To address such a problem, a straightforward solution is checking the package similarities of all the active functions, and assigning the functions that share more packages to the same

node. However, it is not always a good solution, as the functions sharing many packages may not have idle containers.

To resolve the problems above, we propose a function balancing strategy based on the statistics of zygote containers and the available resources $\mathbb{U}_n = \{U_{CPU}, U_{IO}, U_{net}, \dots\}_n$ on every node. The function balancer is implemented on the head node of the cluster, to obtain the statistics from all the nodes. For a function f_B that requires package pkg_a and pkg_b , if it fails to find a zygote container during its invocation on a node, its future invocations should be redirected to another node with potential zygote images.

To this end, Pagurus runs the sharing-aware function balancer on the head node based on the resource usage \mathbb{U}_n , and the similarity between the redirected f_A and functions with idle containers on node n . Let $\mathbb{N} = \{n | \max \mathbb{U}_n \leq T_{res}\}$ represent the set of nodes where the resource utilization is under the threshold T_{res} (80% by default). The head node will select a new node with the most zygote containers from \mathbb{N} and inform the API gateway accordingly. After that, the queries of f_B will be routed to this new node.

8 Evaluation of Pagurus

In this section, we evaluate Pagurus in reducing the cold startups and end-to-end latencies when a function does not have warm containers. Then, we evaluate Pagurus by a large-scale evaluation with Azure trace. After that, we show the integration with other techniques and overhead.

8.1 Experimental Setup

We use 10 best-practice applications with the most GitHub stars from Amazon AWS samples as the benchmarks [6]. We use these applications for revealing the performance of Pagurus for real applications. Experiments with small scale benchmarks in serverless benchmark suites, e.g., FaaS-Profiler [49] and ServerlessBench [61] show similar results. We run the benchmarks on a 6-node cluster. A node generates function invocations, and the other 5 nodes serve invocations. Table 1 shows the configurations of each node.

Pagurus does not rely on the function invocation arrival distribution. In Section 8.2-8.3, we send queries to each application following a Poisson distribution by randomly sampling λ between 0 and 5 queries per second. We co-locate all the benchmarks, and run 20 tests with different samples to avoid randomness. More experiments are done with the Pareto distribution-based invocation pattern of the Azure serverless trace in Section 8.4.

We compare Pagurus, prewarm-disabled OpenWhisk, prewarm-enabled OpenWhisk (OpenWhisk-Prewarm), and SOCK [46]. SOCK also prewarms containers by dynamically updating packages in the prewarmed containers to alleviate cold startups. When a function obtains a prewarmed

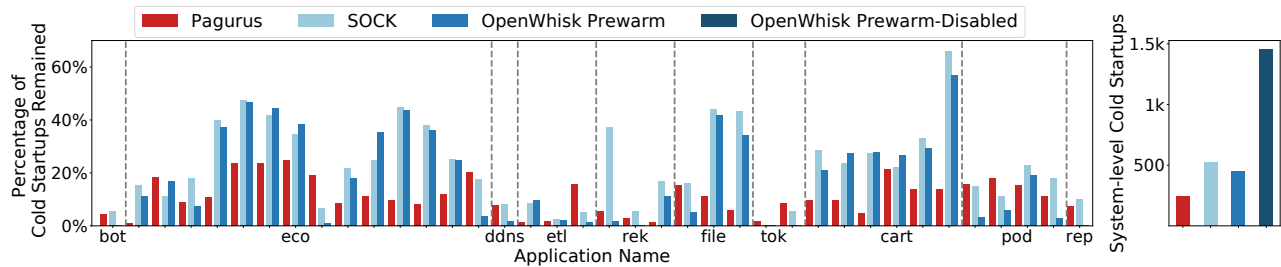


Figure 10: Remained cold startups of OpenWhisk-Prewarm, SOCK and Pagurus, compared with prewarm-disabled OpenWhisk. The left figure shows the remained cold startups (all bars are normalized to the number of cold startups of OpenWhisk Prewarm-Disabled) of each function, while the right figure shows the remained cold startups of all functions in the system.

Table 1: Hardware, software, and benchmark setups

	Configuration
Node	CPU: Intel Xeon(Ice Lake) Platinum 8369B @3.5GHz Cores: 8, DRAM: 16GB, Disk: 100GB SSD (3000 IOPS)
Software	Operating system: Linux with kernel 4.15.7, Docker: 20.10.6 Nginx version: nginx/1.10.3, Database: Couchdb:3.1.1 runc version: 1.0.0-rc93, containerd version: 1.4.4
Container	Container runtime: Python-3.7.0, Linux with kernel 4.15.7 Resource limit and Lifetime: 1-core with 256MB, 600s Function container limit: 10 for each function on each node prewarm pool size in OpenWhisk: 2 on each node
Benchmarks (38 functions in 10 AWS Lambda best practice applications)	serverless-e-commerce-platform (eco), etl-orchestrator (etl) cost-explorer-report (rep), serverless-tokenization (tok) transcribe-comprehend-podcast (pod), serverless-chatbot (bot) serverless-shopping-cart (cart), refarch-fileprocessing (file) finding-missing-persons-using-rekognition (rek), ddns

container, SOCK and OpenWhisk-Prewarm copy the missing packages into the prewarmed container for its invocation.

8.2 Alleviating Container Cold Startups

Figure 10 shows the percentages of the cold startups not eliminated by Pagurus, SOCK, and OpenWhisk-Prewarm, compared with prewarm disabled OpenWhisk. On average, Pagurus alleviate 83.1% of the cold startups, while OpenWhisk-Prewarm and SOCK alleviate 68.9% and 64.4% of that. Meanwhile, as Pagurus does not need to prewarm containers, there is no extra memory overhead introduced by the prewarm container pool with Pagurus.

As observed, SOCK and OpenWhisk-Prewarm only reduce a small percentage of cold startups for many functions (e.g., functions of *eco* and *cart*). This is because the functions tend to contend for the limited prewarmed containers. On the contrary, Pagurus alleviates the cold startups by forking other functions' zygote containers, without trapping in the same dilemmas. We can also find that Pagurus alleviates slightly fewer container cold startups for several functions, compared with SOCK and OpenWhisk-Prewarm. This is because these functions have low cold startup frequency by default, and SF-WRS policy does not tend to re-pack them into zygotes for achieving higher system-level cold startup alle-

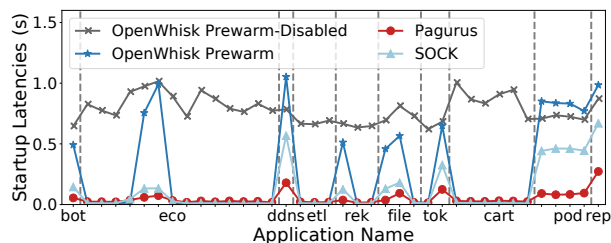


Figure 11: The latency of starting up a prewarmed container in OpenWhisk-Prewarm and SOCK, and the latency of forking a zygote container in Pagurus.

viation. The percentage looks large when the original cold startup frequency is low, even if a small number of cold container startup is not eliminated.

In our experiment, 27.8% of the warm containers are turned into zygote containers, then are forked by other functions. When a function does not have a warm container for its queries, 60.5% of the obtained containers are forked.

8.3 Reducing Startup and E2E Latency

Figure 11 shows the latencies of starting a container from a prewarmed one (OpenWhisk-Prewarm and SOCK), and forking a zygote container (Pagurus). As observed, all the benchmarks have the shortest startup latencies with Pagurus.

If a function invocation is hosted by a helper container with Pagurus, the packages are ready beforehand, and only the user-specific code initialization is needed. Pagurus is able to fork a zygote container in 11ms, and completes the code initialization in 5ms.

With OpenWhisk-Prewarm, starting from prewarmed containers takes longer than directly cold startup a container from the image (e.g., functions in *ddns*, *pod*, and *rep*). SOCK reduces the latency of the prewarm startup leveraging the packages cached with higher benefit-to-cost.

Figure 12 shows the average end-to-end latency of each function normalized to its latency with prewarm-disabled

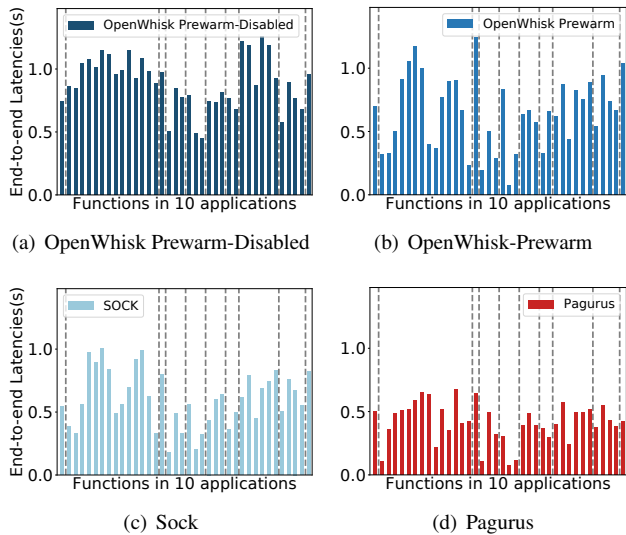


Figure 12: The end-to-end latencies of Prewarm-Disabled OpenWhisk, OpenWhisk Prewarm, SOCK and Pagurus.

OpenWhisk. Pagurus reduces the end-to-end latency of the benchmark functions by 475ms and 479ms on average, while OpenWhisk-Prewarm and SOCK reduce the end-to-end latency by 237ms and 286ms.

By mounting packages beforehand with privilege control, zygote containers can better reduce the startup latency, thus the end-to-end latency.

8.4 Large-scale Evaluation with Azure Trace

In this subsection, we evaluate Pagurus by replaying the Azure serverless trace [50] on a 31-node cluster. The software and hardware configuration of each node is the same as Table 1. We use all the 40,000 functions from the *day07* trace [12], generate function invocations, and randomly route the invocations to the 30 nodes.

The Package similarity in Pagurus is used to shrink the searching space for identifying to-be-helped candidates and reducing the re-packing overhead. However, the Azure trace does not provide package information for the functions, but only the function duration and invocation arrival time. Lacking the package information, it is impossible to evaluate the similarity-filtered WRS selection policy in Pagurus, nor OpenWhisk-Prewarm or SOCK. With no package information and similarity-filtered re-packing for Azure trace, Pagurus identifies to-be-helped candidates based on the basic WRS policy. We therefore only compare similarity-disabled Pagurus, with the prewarm-disabled OpenWhisk for the large-scale evaluation, to show the effectiveness of alleviating cold startups by simply replacing idle containers with zygotes.

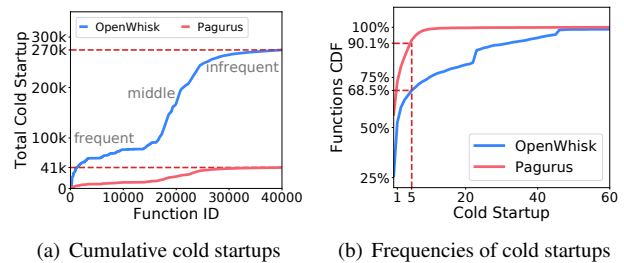


Figure 13: The effect of alleviating cold startup, and the CDF that functions suffer from different cold startup frequencies.

8.4.1 Alleviating Cold Startup

We report two user experience-related metrics in this experiment. First, how many cold startups are alleviated by Pagurus? Second, how many functions seldom experience cold startups (e.g., one cold startup) in one day with Pagurus?

Figure 13(a) shows the total number of cold startups with Pagurus and prewarm-disabled OpenWhisk (denoted by “OpenWhisk” for short in this subsection). In the figure, the functions are sorted in the descending order of their invocation frequencies. The smaller the function ID, the more frequent the function is invoked. As shown in the figure, Pagurus reduces the number of cold startups by 84.6%. We can also find that OpenWhisk results in the frequent cold startup for the functions of middle-popularity. It is because the warm containers of these middle-popularity functions tend to be recycled due to the relatively low invocation frequencies. Pagurus efficiently alleviates the cold startups for the middle-popularity functions through zygote containers.

Figure 13(b) shows the cumulative distribution of the functions with different container cold startup frequencies. As observed, 73.4% and 52.1% of all functions experience cold startup less than once in a day with Pagurus and OpenWhisk, respectively. Meanwhile, 90.1% of the functions experience cold startups less than 5 times daily with Pagurus. In the figure, sudden jumps happen around 24 and 48 cold startups for OpenWhisk. The jumps are caused by functions with a 1-hour or 30-minutes trigger in the trace.

Pagurus effectively alleviates the cold container startup, especially for middle-popularity and low-popularity functions in production. It greatly improves the user experience.

8.4.2 Reducing Tail Latency

Figure 14 shows the 95%-ile latencies of the 40,000 functions with Pagurus and OpenWhisk. The left y-axis shows the 95%-ile latencies of functions with Pagurus, and the right y-axis shows that with OpenWhisk normalized to the former. OpenWhisk results in longer 95%-ile latencies for most functions than Pagurus (the right y-axis is larger than 1).

We can also observe that popular functions (functions

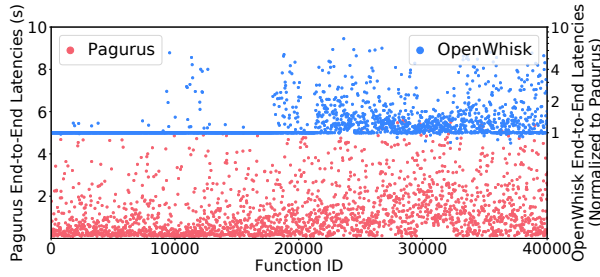


Figure 14: The end-to-end 95%-ile latency of 40,000 functions with Pagurus and OpenWhisk.

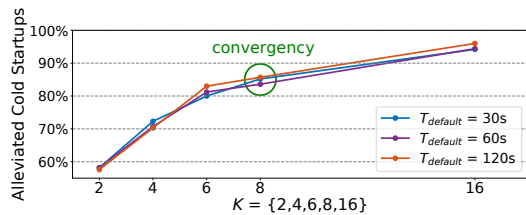


Figure 15: The convergency and the percentages of cold startups alleviated by Pagurus with different $T_{default}$.

with ID smaller than 15,000) have similar 95%-ile latency with OpenWhisk and Pagurus. This is because the slowest 95%-ile invocations of these functions still experience warm startup, as these functions are frequently invoked. For the middle-popularity and low-popularity functions, their 95%-ile latencies are the latency of the function invocation that suffers from the cold container startup with OpenWhisk.

8.4.3 Impacts of Hyperparameters

In this experiment, we evaluate the impact of $T_{default}$ (the value is set as 60s by default), and the number of to-be-helped functions K , on Pagurus. Figure 15 shows the percentages of cold startups alleviated by Pagurus with different $T_{default}$ and different K . As observed, the performance of Pagurus is stable when $T_{default}$ varies. The performance of Pagurus is not sensitive to $T_{default}$.

We also find the number of to-be-helped functions, K , gradually converges to 8. Moreover, the appropriate value of K is not affected by $T_{default}$. For Azure workloads, Pagurus should generate a zygote for 8 to-be-helped functions on average. It is consistent with the calculated one in Equation 6. We also measure the impact of larger prewarm pool size in OpenWhisk, and find that the improvement becomes marginal but with significant resource waste.

8.5 Integrating with Orthogonal Techniques

The decoupled hierarchy design of Pagurus provides easy-to-use APIs for container orchestrators. Pagurus can be integrated with prior work on speeding up the cold startup.

Table 2: Overheads of the components in Pagurus

Sources	Type	Overheads (each node)
Intra-container manager	CPU overhead	0.345 core
	Memory overhead	228MB
	Storage overhead	485MB for each zygote image
Inter-function scheduler	CPU overhead	0.66 core (re-packing included)
	Memory overhead	315MB

Pagurus brings shorter end-to-end latency when it is integrated with Checkpoint/Restore [7] (denoted by C/R) and Catalyzer [28], respectively. With C/R, a container is recovered from a checkpoint image. With Catalyzer, more data are already loaded in the image stored in memory. By replaying the evaluation, we find that C/R+Pagurus reduces the cold startup time of the benchmarks by 78.9% on average compared with C/R; Catalyzer+Pagurus reduces the cold startup time by 15.1% on average compared with Catalyzer. Even if no appropriate forked zygote container returns, Pagurus does not slow down the container startup.

8.6 Overheads of Pagurus Components

In Pagurus, *packing zygote images, generating zygote containers from the images, and determining the to-be-helped functions* for each function introduce runtime overhead.

According to our measurement, each container in Pagurus uses smaller memory on average than OpenWhisk. The reduction originates from the design of the zygote container. Although packages are pre-installed in zygote containers, they are imported into memory only when a zygote container is forked. On the contrary, warm containers (private containers) always keep the packages in memory for low latency. The reduction of memory usage is not affected by the number of to-be-helped functions.

Table 2 shows the CPU, memory, and storage overhead caused by the intra-container managers and the inter-function schedulers when replaying the Azure trace. As reported, less than one core is required to run all the intra-container managers and the inter-function scheduler on a node. If fewer functions are executed on a node, the overhead will be smaller.

9 Conclusion

Pagurus alleviates cold startups with inter-function container sharing rather than popular prewarm-based methods. It comprises an intra-function manager for idle container identification and management, an inter-function scheduler for safe container scheduling, and a sharing-aware function balancer for resource-aware workload balancing. Our experimental results based on both real system benchmarks and Azure trace show that Pagurus significantly alleviates the cold container startup. The cold startup latency is reduced from hundreds of milliseconds to 16ms if Pagurus alleviates it.

Acknowledgment

We would thank our anonymous reviewers and shepherd, for their helpful comments and suggestions. This work is partially sponsored by the National Natural Science Foundation of China (62022057, 61832006, 61872240), Open Research Projects of Zhejiang Lab (2021KE0AB02), and Shanghai international science and technology collaboration project (21510713600). Quan Chen and Minyi Guo are the corresponding authors.

References

- [1] Alibaba function compute. <https://alibabacloud.com/product/function-compute>, 2021.
- [2] Annotations on openwhisk assets. <https://github.com/apache/openwhisk/blob/90c20a847b9a70b43e316fd89a0a15ae2ee39cc4/docs/annotations.md>, 2021.
- [3] Apache openwhisk. <https://openwhisk.apache.org>, 2021.
- [4] Authentication and authorization in azure functions. <https://docs.microsoft.com/en-us/azure/app-service/overview-authentication-authorization>, 2021.
- [5] Aws lambda. <https://aws.amazon.com/lambda/>, 2021.
- [6] Aws samples. <https://github.com/aws-samples/>, 2021.
- [7] Checkpoint/restore. <https://criu.org/Checkpoint/Restore>, 2021.
- [8] Execute mode in fission. <https://docs.fission.io/docs/usage/executor/>, 2021.
- [9] Fission workflows: Fast, reliable and lightweight function composition for serverless functions. <https://docs.fission.io/docs/workflows/>, 2021.
- [10] Function identity in google cloud functions. <https://cloud.google.com/functions/docs/securing/function-identity>, 2021.
- [11] Google cloud functions. <https://cloud.google.com/functions>, 2021.
- [12] Microsoft azure functions. <https://azure.microsoft.com/en-us/services/functions>, 2021.
- [13] Nginx. <https://www.nginx.com/>, 2021.
- [14] Prewarm in apache openwhisk. <https://github.com/apache/openwhisk/blob/master/docs/actions-python.md>, 2021.
- [15] Prewarm in azure functions. <https://docs.microsoft.com/en-us/azure/azure-functions/functions-premium-plan>, 2021.
- [16] Troubleshooting aws lambda identity and access. https://docs.aws.amazon.com/lambda/latest/dg/security_iam_troubleshoot.html, 2021.
- [17] RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022. USENIX Association.
- [18] Mainak Adhikari, Tarachand Amgoth, and Satish Narayana Srirama. A survey on scheduling strategies for workflows in cloud environment and emerging trends. *ACM Comput. Surv.*, 52(4):68:1–68:36, 2019.
- [19] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, and Klaus Satzke. SAND: Towards high-performance serverless computing. In *ATC*, pages 923–935, 2018.
- [20] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, and Dimitris Skourtis. Improving docker registry design based on production workload analysis. In Nitin Agrawal and Raju Rangaswami, editors, *16th USENIX Conference on File and Storage Technologies, FAST 2018, Oakland, CA, USA, February 12-15, 2018*, pages 265–278. USENIX Association, 2018.
- [21] Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. Serverless computing: Current trends and open problems. In Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya, editors, *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.
- [22] Kahina Bessai, Samir Youcef, Ammar Oulamara, Claude Godart, and Selmin Nurcan. Bi-criteria workflow tasks allocation and scheduling in cloud computing environments. In *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*, pages 638–645. IEEE Computer Society, 2012.
- [23] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In Haryadi S. Gunawi and Benjamin Reed,

editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 645–650. USENIX Association, 2018.

- [24] Eric A. Brewer. Kubernetes and the path to cloud native. In Shahram Ghandeharizadeh, Sumita Barahmand, Magdalena Balazinska, and Michael J. Freedman, editors, *SoCC*, page 167. ACM, 2015.
- [25] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.*, 25(6):599–616, 2009.
- [26] James Cadden, Thomas Unger, Yara Awad, and Han Dong. SEUSS: skip redundant paths to make serverless fast. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 32:1–32:15. ACM, 2020.
- [27] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarini. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In Dilma Da Silva and Rüdiger Kapitza, editors, *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*, pages 356–370. ACM, 2020.
- [28] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 467–481. ACM, 2020.
- [29] Pavlos S. Efraimidis and Paul G. Spirakis. Weighted random sampling with a reservoir. *Inf. Process. Lett.*, 97(5):181–185, 2006.
- [30] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, Xin Peng, Wenli Zheng, and Minyi Guo. Qos-aware and resource efficient microservice deployment in cloud-edge continuum. In *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*, pages 932–941. IEEE, 2021.
- [31] Alexander Fuerst and Prateek Sharma. Faas-cache: keeping serverless computing alive with greedy-dual caching. In Tim Sherwood, Emery Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 386–400. ACM, 2021.
- [32] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In Angela Demke Brown and Florentina I. Popovici, editors, *FAST*, pages 181–195. USENIX Association, 2016.
- [33] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, and Johann Schleier-Smith. Serverless computing: One step forward, two steps back. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.
- [34] Scott Hendrickson, Stephen Sturdevant, Edward Oakes, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. *log in Usenix Mag.*, 41(4), 2016.
- [35] M. Reza HoseinyFarahabady, Albert Y. Zomaya, and Zahir Tari. A model predictive controller for managing qos enforcements and microarchitecture-level interferences in a lambda platform. *IEEE Trans. Parallel Distributed Syst.*, 29(7):1442–1455, 2018.
- [36] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: a berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.
- [37] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*, pages 158–164. ACM, 2019.
- [38] Young Ki Kim, M. Reza HoseinyFarahabady, Young Choon Lee, and Albert Y. Zomaya. Automated fine-grained CPU cap control in serverless computing platform. *IEEE Trans. Parallel Distributed Syst.*, 31(10):2289–2301, 2020.
- [39] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. DADI: block-level image service for agile and elastic application deployment. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 727–740. USENIX Association, 2020.
- [40] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, BingSheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Comput. Surv.*, dec 2021. Just Accepted.

- [41] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. Faasflow: enable efficient workflow execution for function-as-a-service. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 782–796. ACM, 2022.
- [42] Nima Mahmoudi, Changyuan Lin, Hamzeh Khazaei, and Marin Litoiu. Optimizing serverless computing: introducing an adaptive function placement algorithm. In Tima Pakfetrat, Guy-Vincent Jourdan, Kostas Kontogiannis, and Robert F. Enenkel, editors, *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, CASCON 2019, Markham, Ontario, Canada, November 4-6, 2019*, pages 203–213. ACM, 2019.
- [43] Sean McDaniel, Stephen Herbein, and Michela Taufer. A two-tiered approach to I/O quality of service in docker containers. In *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*, pages 490–491. IEEE Computer Society, 2015.
- [44] M. Garrett McGrath and Paul R. Brenner. Serverless computing: Design, implementation, and performance. In Aibek Musaev, João Eduardo Ferreira, and Teruo Higashino, editors, *ICDCS Workshop*, pages 405–410. IEEE Computer Society, 2017.
- [45] Anup Mohan, Harshad S. Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In Christina Delimitrou and Dan R. K. Ports, editors, *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, Renton, WA, USA, July 8, 2019*. USENIX Association, 2019.
- [46] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {SOCK}: Rapid task provisioning with serverless-optimized containers. In *{USENIX} Annual Technical Conference (ATC)*, pages 57–70, 2018.
- [47] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In Jay R. Lorch and Minlan Yu, editors, *NSDI*, pages 193–206. USENIX Association, 2019.
- [48] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 753–767. ACM, 2022.
- [49] Mohammad Shahradsad, Jonathan Balkind, and David Wentzlauff. Architectural implications of function-as-a-service computing. In *Micro*, pages 1063–1075. ACM, 2019.
- [50] Mohammad Shahradsad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Riccardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In Ada Gavrilovska and Erez Zadok, editors, *ATC*, pages 205–218. USENIX Association, 2020.
- [51] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra. *CoRR*, abs/1810.09679, 2018.
- [52] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *ASPLOS*, pages 121–135, 2019.
- [53] Eli Tilevich and Hanspeter Mössenböck, editors. *International Conference on Managed Languages & Run-times*. ACM, 2018.
- [54] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In Tim Sherwood, Emery Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 559–572. ACM, 2021.
- [55] Ranjan Sarpangala Venkatesh, Till Smejkal, Dejan S. Milojicic, and Ada Gavrilovska. Fast in-memory criu for docker containers. In *International Symposium on Memory Systems*, pages 53–65, New York, NY, USA, 2019. Association for Computing Machinery.
- [56] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In Andrew Herbert and Kenneth P. Birman, editors, *SOSP*, pages 148–162. ACM, 2005.

- [57] T. Wagner. Understanding container reuse in aws lambda. <https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/>, 2021.
- [58] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Re-playable execution optimized for page sharing for a managed runtime environment. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Eurosys*, pages 39:1–39:16. ACM, 2019.
- [59] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. Peeking behind the curtains of serverless platforms. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 133–146. USENIX Association, 2018.
- [60] Zhengjun Xu, Haitao Zhang, Xin Geng, Qiong Wu, and Huadong Ma. Adaptive function launching acceleration in serverless computing platforms. In *25th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2019, Tianjin, China, December 4-6, 2019*, pages 9–16. IEEE, 2019.
- [61] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi, editors, *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, pages 30–44. ACM, 2020.

A Artifact Appendix

A.1 Abstract

Our artifact includes the prototype implementation of Zygote-based mechanism in Pagurus, 10 masked applications (including 38 functions) benchmarks, and mapped functions from Azure Trace. The artifact provides experiment workflow scripts to perform the measurement.

A.2 Artifact Check-list (Meta-information)

- **Run-time Environment:** Ubuntu 18.04, Docker 20.10.6, CouchDB 3.2.2 and Python are required.
- **Data set:** The artifact uses 10 masked application benchmarks from AWS samples and Azure traces.
- **Execution workflows:** For reproducing our paper’s results, we provide the corresponding scripts for each evaluation section (from Section 8.2 to Section 8.4) to send queries, collect the execution metrics, and draw the comparison plots.
- **Time needed to complete:** see the instruction of each Exp.
- **Publicly available:** <https://github.com/lzjzx1122/Pagurus>
- **Code Licenses:** Apache-2.0 license

A.3 Hardware and Software Dependencies

- **Hardware:** The hardware is configured by {CPU: Intel Xeon(Ice Lake) Platinum 8369B @3.5GHz, Cores: 8, DRAM: 16GB, Disk: 200GB SSD with 4200 IOPS.}
- **Software environment:** Operating system: {Linux with kernel 4.15.0, Docker: {20.10.15}, Container runtime: {Python-3.6.9, Linux with kernel 4.15.7}, Nginx version: {nginx/1.10.3}, Database: {Couchdb with version 3.2.2}, runc version: {1.0.0-rc93}, containerd version: {1.4.4}, and Pagurus. Detailed software dependencies are all listed and scripted in the artifact.

A.4 How to Access and Install

GitHub link: <https://github.com/lzjzx1122/Pagurus>. Clone the GitHub repository and then run the quick setup script to deploy Pagurus.

A.5 Experiment and Expected Results

A.5.1 AWS applications (Section 8.2 and 8.3)

Under the path `Pagurus/aws/trace`, there are 18 different sampling test results for AWS applications, and the expected test results for each sampling are stored in the directory `aws/expected_result`. You can directly run `aws/plot.py` to generate the plots, or replay the trace using a testing script to run the AWS experiment:

```
$ python3 aws/run_experiments.py 1
```

Experiment customization: The above script performs the 1st sampling test with Openwhisk, Pagurus, OpenWhisk-Prewarm and SOCK. Other sampling tests can also be performed by changing "1" to other test numbers. To fully reproduce our result, it additionally takes at least 160-hours (20 tests with different sampling parameters) to generate the execution logs for 4 platforms. To ensure that the zygote repackaging mechanism and prewarm works efficiently, the running time should be at least 2-hours for both 4 platforms (8 hours for each test number).

After running the sampling tests under four platforms, the script will generate the results under `aws/result`. Run the following script to generate three .csv files:

```
$ python3 aws/summary_from_results.py
```

- `cold_start.csv` shows the remained cold startups of OpenWhisk, SOCK and Pagurus, compared with prewarm-disabled OpenWhisk (Figure 11).
- `startup_time.csv` shows latencies of starting up containers in prewarm-disabled OpenWhisk, OpenWhisk and SOCK, respectively. It also contains latencies of forking zygote containers in Pagurus (Figure 12).

- `e2e_latency.csv` shows e2e latencies of benchmarks in Pagurus, SOCK, OpenWhisk, and prewarm-disabled OpenWhisk, respectively. (Figure 13).

Then run the following script to generate the plots:

```
$ python3 aws/plot_from_results.py
```

A.5.2 Azure Trace mapping (Section 8.4)

In the Azure trace experiment (Day07), invoke more than 40,000 functions will take 24 hours and more than 800 vCPUs by default. To reduce the computation resources needed, we randomly select about 4,000 functions to generate several small-scale Azure traces under the path `Pagurus/azure/trace`. The functions in each small-scale trace are all different from each other. A larger trace with more functions can be replayed if only more computation resources or nodes are provided.

Considering that the experiment will take about 24 hours, we already pre-run each small-scale Azure trace and save their execution results. You can directly run `azure/plot.py` to generate the plots, or replay the trace using the following script:

```
$ python3 azure/run_experients.py 1
```

Experiment customization: Each small-scale Azure trace can be replayed by changing "1" to other trace numbers. The script will replay the trace in Openwhisk and Pagurus, respectively, and then generate results under `azure/result`. Run the following script to generate two `.csv` files:

```
$ python3 azure/summary_from_results.py
```

- `cold_start.csv` shows the remained cold startups of OpenWhisk and Pagurus, respectively (Figure 14).
- `e2e_latency.csv` shows end-to-end 95%-ile latencies of benchmarks in Openwhisk and Pagurus, respectively (Figure 15).

Then run the following script to generate the plots:

```
$ python3 azure/plot_from_results.py
```

RRC: Responsive Replicated Containers

Diyu Zhou
UCLA and EPFL

Yuval Tamir
UCLA

Abstract

Replication is the basic mechanism for providing application-transparent reliability through fault tolerance. The design and implementation of replication mechanisms is particularly challenging for general multithreaded services, where high latency overhead is not acceptable. Most of the existing replication mechanisms fail to meet this challenge.

RRC is a fully-operational fault tolerance mechanism for multiprocessor workloads, based on container replication. It minimizes the latency overhead during normal operation by addressing two key sources of this overhead: (1) it decouples the latency overhead from checkpointing frequency using a hybrid of checkpointing and replay, and (2) it minimizes the pause time for checkpointing by forking a clone of the container to be checkpointed, thus allowing execution to proceed in parallel with checkpointing. The fact that *RRC* is based on checkpointing makes it inherently less vulnerable to data races than active replication. In addition, *RRC* includes mechanisms that further reduce the vulnerability to data races, resulting in high recovery rates, as long as the rate of manifested data races is low. The evaluation includes measurement of the recovery rate and recovery latency based on thousands of fault injections. On average, *RRC* delays responses to clients by less than 400 μ s and recovers in less than 1s. The average pause latency is less than 3.3ms. For a set of eight real-world benchmarks, if data races are eliminated, the performance overhead of *RRC* is under 48%.

1 Introduction

For many applications hosted in data centers, high reliability is a key requirement, demanding fault tolerance. The key desirable properties of a fault tolerance mechanism, especially for server applications, are: A) Low throughput and latency overheads; B) Support for multithreaded applications; and C) Application transparency. Replication, has long been used to implement application-transparent fault tolerance, especially for server applications.

The two main approaches to replication, specifically, duplication, are: (1) high-frequency checkpointing of the primary replica state to a passive backup [33], and (2) active replication, where the primary and backup both execute the application [38, 47]. A key disadvantage of the first approach is that, for consistency between server applications and their clients after failover, outputs must be delayed before being released to the client, typically for tens of milliseconds. Such delays are unacceptable for many server applications.

To support active replication of multiprocessor workloads,

where there are many sources of nondeterminism, active replication is implemented using a leader-follower algorithm. With this algorithm, the outcomes of identified nondeterministic events on the primary, namely synchronization operations and certain system calls, are recorded and sent to the backup. This allows the backup to deterministically replay their outcomes [38, 47]. A disadvantage of this approach is that it is vulnerable to even rare replay failures due to untracked nondeterministic events, such as those caused by data races. Another disadvantage is that, for application with a high rate of synchronization operations, the replay on the backup may be significantly slower than the execution on the primary, resulting in high throughput overhead [38]. This is due to the interaction between thread scheduling by the OS and the need to mirror on the backup the execution on the primary.

This paper presents a fault tolerance scheme, based on container replication, called *RRC* (Responsive Replicated Containers). *RRC* targets server applications and is thus optimized to minimize response latency overhead. *RRC* overcomes the disadvantages of existing approaches using a combination of periodic checkpointing [33, 62] and externally-deterministic replay [29]. The primary sends periodic checkpoints to the passive backup. While executing, the primary logs to the backup the outcomes of nondeterministic events. Upon failure, the backup restores the latest checkpoint and deterministically replays the execution up to the last external output. Hence, external outputs only need to be delayed by the short amount of time it takes to send and commit the relevant portion of the nondeterministic event log to the backup.

RRC minimizes request-reply latency overhead, not only for the average case, but also the tail latency overhead. To that end, *RRC* had to overcome a key challenge, namely, that while the state of the primary is collected for transmission to the backup, execution has to be paused. Even with various optimizations, this latency is often tens of milliseconds, largely due to the cost of retrieving in-kernel state associated with the container, such as the state of open file descriptors [62]. To meet this challenge, *RRC* introduces a new kernel primitive: container fork. For checkpointing, *RRC* pauses the primary container, forks a shadow container, and resumes execution. This results in pause times of less than 3.5ms. The checkpoint is obtained from the shadow container.

RRC decouples the response latency from the checkpointing duration. This enables high performance by allowing the tuning of epoch duration to trade off performance and resource overheads with recovery latency and vulnerability to untracked nondeterministic events. The latter is important

since applications may contain data races (§3, §6.3). *RRC* is focused on dealing with data races that rarely manifest and are thus more likely to remain undetected (§3). Since *RRC* only requires replay during recovery and for the short interval since the last checkpoint, it is inherently more resilient to data races than active replication schemes that rely on replay of the entire execution [38]. Furthermore, *RRC* includes timing adjustment mechanisms that result in a high recovery rate even for applications that include data races, as long as their rate of unsynchronized writes is low (§4.7).

RRC also decouples the performance on the primary from the backup. Thus, unlike active replication schemes [38], the backup is not a performance bottleneck (§6.1).

Replication can be at the level of VMs [27, 33, 35, 53, 58], processes [32, 38, 47], or containers [62]. Containers have advantages over VMs due to smaller memory and storage footprints, faster startup, and avoiding the need to manage updates of multiple VMs [25, 45]. Furthermore, containers are the best fit for mechanisms such as *RRC*. Applying *RRC*'s approach to VMs would be complicated since there would be a need to track and replay nondeterministic events in the kernel. On the other hand, with processes, it is difficult to avoid potential name conflicts (e.g., process IDs) upon failover. While such name conflicts can be solved, the existing container mechanism already solves them efficiently.

The implementation of *RRC* involved developing solutions to implementation challenges that have not been addressed by prior works. The most important of these is dealing with the integration of timer-triggered checkpointing, that is not synchronized with the application, and user-level recording of nondeterministic events (§4.2). *RRC* also efficiently handles the failover of TCP connections through checkpoint restoration, a replay phase, and finally resumption of live execution (§4.3, §4.4). *RRC* is application-transparent and does not require any changes to the application code.

We have implemented a prototype of *RRC* and evaluated its performance and reliability. With 1s epochs, *RRC*'s throughput and average latency overheads were less than 49% and 230 μ s, respectively, for all eight benchmarks. With 100ms epochs, the corresponding overheads were less than 53% and 291 μ s for seven benchmarks, 86% and 264 μ s for the eighth. *RRC* is designed to recover from fail-stop faults. We used thousands of fault injections to validate and evaluate *RRC*'s recovery mechanism. For all eight benchmarks, after data races identified by ThreadSanitizer [6] were resolved, *RRC*'s recovery rate was 100% for 100ms and 1s epochs. Three of the benchmarks originally included data races. For two of these, without any modifications, with 100ms epochs and *RRC*'s timing adjustments, the recovery rate was over 99.1%.

RRC achieves both low response latency overhead and resilience to infrequently-manifested data races. This combination provides a fundamental advance over both Remus-based techniques [33] and active replication [38, 47], respectively. Specifically, we make the following contributions: 1) a

fault tolerance scheme based on container replication, using a unique combination of periodic checkpointing, deterministic replay, and an optimized scheme for failover of network connections; 2) a new system call, container fork, used to minimize tail latency overhead; 3) a replication mechanism with inherent resilience to untracked nondeterministic events, further enhanced by mechanisms that increase recovery success rate in the presence of data races; 4) a thorough evaluation of *RRC* with respect to performance overhead, resource overhead, and recovery rate, demonstrating the lowest reported external output delay compared to competitive mechanisms.

Section 2 presents two key building blocks for *RRC*: NiLiCon [62] and deterministic replay [21, 29, 44, 50, 57]. An overview of *RRC* is presented in §3. *RRC*'s implementation is described in §4, with a focus on key challenges. The experimental setup and evaluation are presented in §5, and §6, respectively. Limitation of *RRC* and of our prototype implementation are described in §7. §8 provides a brief overview of related work.

2 Background

RRC integrates container replication based on periodic checkpointing [33, 62], described in §2.1, and deterministic replay of multithreaded applications, described in §2.2.

2.1 NiLiCon

Remus [33] introduced a practical application-transparent fault tolerance scheme based on VM replication using high-frequency checkpointing. NiLiCon [62] is an implementation of the Remus mechanism for containers. A key challenge faced by NiLiCon is that, compared to VMs, there is much tighter coupling between the container state and the state of the underlying platform. NiLiCon meets this challenge, based on a tool called CRIU (Checkpoint/Restore in User Space) [4], with novel optimizations that significantly reduce overhead. CRIU checkpoints and restores the user-level and kernel-level state of a container, except for disk state. NiLiCon handles disk state by adding system calls to checkpoint and restore the page cache and a modified version of the DRBD module [8]. NiLiCon relies on CRIU to preserve established TCP connections across failover, using a special repair mode of the socket provided by the Linux kernel [18].

2.2 Deterministic Replay on Multiprocessors

Deterministic replay is the reproduction of some original execution in a subsequent execution. During the original execution, the results of nondeterministic events/actions are recorded in a log. This log is used in the subsequent execution [29]. With a uniprocessor, nondeterministic events include: asynchronous events, such as interrupts; system calls, such as *gettimeofday()*; and inputs from the external world.

With shared-memory multiprocessors, there is a higher frequency of nondeterministic events related to the order of

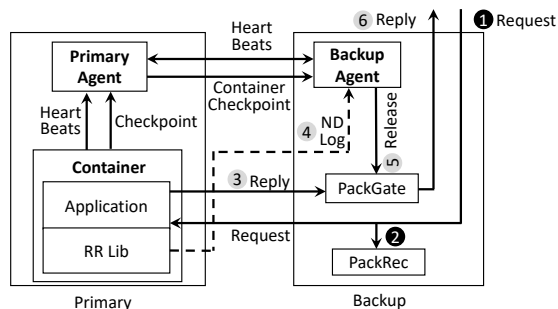


Figure 1: Architecture and workflow of *RRC*.

accesses to the same memory location by different processors. For such systems, a common approach is to support deterministic replay only for programs that are data-race-free [49]. For such programs, as long as the results of synchronization operations are deterministically replayed, the ordering of shared memory accesses are preserved. The recording of nondeterministic events can occur at different levels: hardware [40,59], hypervisor [36,37,41], OS [39,43], or library [49,54]. Without dedicated hardware support, it is advantageous to record the events at the user level, thus avoiding the overhead for entering the kernel or hypervisor [44].

To support seamless failover with replication, it is sufficient to provide *externally deterministic replay* [44]. This means that, with respect to what is visible to external clients, the replayed execution is identical to the original execution. Furthermore, the internal state at the end of replay must be a state that corresponds to a possible original execution that could result in the same external behavior. This latter requirement is needed so that the replayed execution can transition to consistent live execution at the end of the replay phase.

3 Overview of *RRC*

RRC provides fault tolerance by maintaining a primary-backup pair with an inactive backup that takes over when the primary fails. Execution on the primary is divided into epochs and the primary state is checkpointed to an inactive backup at the end of each epoch [33,62]. Upon failure of the primary, the backup begins execution from the last primary checkpoint and then deterministically replays the primary's execution of its last partial epoch, up to the last external output. The backup then proceeds with live execution. To support the backup's deterministic replay, *RRC* ensures that, before an external output is released, the backup has the log of nondeterministic events on the primary since the last checkpoint. Thus, external outputs are delayed only by the time it takes to commit the relevant last portion of the log to the backup.

Figure 1 shows the overall architecture of *RRC*. The primary records nondeterministic events: operations on locks and nondeterministic system calls. The record and replay are done at the user level, by instrumentation of glibc source code. When the primary executes, the instrumented code invokes functions in a dedicated RR (Record and Replay) library that create logs used for replay. There is a separate log for each

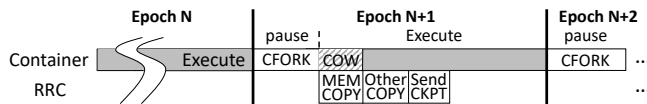


Figure 2: Timeline of an epoch on the primary replica.

lock. For each thread, there is a log of the nondeterministic system calls it invoked, with their arguments and return values. Details are presented in §4.1.

Figure 1 shows the processing of requests and replies for server applications. (1) Client requests are sent to the backup. (2) To support TCP failover, the backup records incoming packets and forwards them to the primary. (3) Replies from the primary are forwarded to the backup and blocked by the *PackGate* queueing discipline kernel module. (4) The primary sends the nondeterministic event log to the backup. (5) Upon receiving the log, *PackGate* releases the corresponding replies.

Figure 2 shows a timeline of each epoch on the primary replica. First, the container is paused and a container fork is performed. Execution is then resumed. The first write to a page results in a Copy On Write (COW) so that the state of the forked shadow container is maintained. Concurrently, the pages modified since the last checkpoint are copied to a staging buffer (*MEM COPY*). Once this copy is completed, the original container ceases to perform the COW operations. A container checkpoint includes in-kernel state associated with the container, such as the state of open file descriptors [62]. This state is obtained from the shadow container and written to the staging buffer (*Other COPY*). The entire checkpoint is then sent from the primary to the backup.

RRC is based on having the ability to identify all sources of nondeterminism that are potentially externally visible, record their outcomes, and replay them when needed. Thus, unsynchronized accesses to shared memory during the epoch in which the primary fails may cause replay on the backup to fail to correctly reproduce the primary's execution, leading the backup to proactively terminate. This implies that applications are expected to be free of data races. However, not all multithreaded programs meet this expectation. Furthermore, precise race detection is NP-hard [48]. Hence, it is not possible to ensure that all data races are detected and eliminated. Fortunately, frequently-manifested data races are detectable using tools such as ThreadSanitizer [6]. Hence, only rarely-manifested data races are likely to remain in applications.

Since *RRC* only requires replay of short intervals (up to one epoch), it is inherently more tolerant to rarely-manifested data races than schemes that rely on accurate replay of the entire execution [38]. As an addition to this inherent advantage of *RRC*, *RRC* includes optional mechanisms that significantly increase the probability of correct recovery despite data races, as long as the manifestation rate is low (§4.7). During execution on the primary, these mechanisms record the order and timing of returns from nondeterministic system calls by *all* the threads. During replay, the recorded order and relative

timing are enforced.

If the primary fails, network connections must be maintained and migrated to the backup [19, 20, 22, 60]. Like CoRAL [19, 20], requests are routed through backup by advertising the service IP address in the backup. Unlike FT-TCP [22, 60] or CoRAL, replies are also routed through the backup, resulting in lower latency (§4.3).

As with most other state replication work [33, 53, 58, 62], *RRC* assumes fail-stop faults. Either the primary or the backup may fail. Heartbeats are exchanged between the primary and backup so failures are detected as missing heartbeats. Thus, *RRC* relies in the synchrony assumption [34] with respect to both the hosts and the network. If the backup fails, the primary configures its network, advertises the service IP address, and communicates with the clients directly. To maintain redundancy, a new backup needs to be instantiated and take over the service IP address.

4 Implementation

This section presents the implementation of *RRC*, focusing on the mechanisms used to overcome key challenges. *RRC* is implemented mostly at the user level but also includes small modifications to the kernel. At the user level, the implementation includes: agent processes on the primary and backup hosts that run outside the replicated container; a special version of the glibc library (that includes Pthreads), where some of the functions are instrumented (wrapped), used by the application in the container; and a dedicated RR (record and replay) library, that provides functions that actually perform the record and replay of nondeterministic events, used by the application in the container.

The kernel modifications include: an ability to record and enforce the order of access to key data structures (§4.1); support for a few variables shared between the kernel and RR library, used to coordinate checkpointing with record and replay (§4.2); a new queueing discipline kernel module used to pause and release network traffic (§4.3); and container fork (§4.6).

In the rest of this section, §4.1 presents the basic record and replay scheme. §4.2 deals with the challenge of integrating checkpointing with record and replay. §4.3 presents the handling of network traffic. The transition from replay to live execution is discussed in §4.4. The performance-critical operation of transmitting the nondeterministic event log to the backup is explained in §4.5. Container fork is presented in §4.6. §4.7 presents our best-effort mechanism for increasing the probability of correct replay in the presence of infrequently-manifested data races.

4.1 Nondeterministic Events Record/Replay

To minimize overhead and implementation complexity, *RRC* records synchronization operations and system calls at the user level. This is done by code added in glibc before (*before*

hook) and after (*after hook*) the original code. Recording is done in the after hook, replay is in the before hook.

For each lock, there is a log of lock operations in the order of returns from those operations. The log entry includes the ID of the invoking thread and the return value. The return values are recorded to handle the *trylock* variants as well as errors. During replay, synchronization operations must actually be performed in order to properly enforce the correct semantics. For each lock, the ordering of successful lock acquires is enforced. Since there is no need to enforce ordering among different locks, it is sufficient to maintain a separate log for each lock.

For each thread, there is a log of invoked system calls. The log entry includes the parameters and return values. During replay, the recorded parameters are used to detect divergence (replay failure). For some functions, such as *gettimeofday()*, replay does not involve the execution of the function and the recorded return values are returned. However, as discussed in §4.4, functions, such as *open()*, that involve the manipulation of kernel state, are actually executed during replay.

There can be dependencies among system calls, even if they are invoked by different threads. For example, this is the case for system calls whose execution involve writes and reads from kernel data structures, such as the file descriptor table. Hence, simply maintaining a separate log for each thread is not sufficient. To handle such cases, the kernel was modified to maintain an access sequence number for each such shared kernel resource. Each thread registers the address of a per-thread variable with the kernel. When the thread executes a system call accessing a shared resource, the kernel increments the sequence number and copies its value to the registered address. At the user level, this sequence number is attached to the corresponding system call log entry. During replay, the before and after hooks enforces the recorded execution order.

4.2 Integrating Checkpointing with Record/Replay

Checkpointing is triggered by a timer external to the container [62], and is thus not synchronized with the recording of nondeterministic events on the primary. This has the potential of resulting in a checkpoint and log contents on the backup from which correct replay cannot proceed. One example is that the checkpoint may include a thread in the middle of executing code in the RR library, resulting in the backup, during replay, attempting to send the nondeterministic event log to the backup. A second example is that there may be ambiguity at the backup as to whether a particular system call, such as *open()*, was executed after the checkpoint and thus needs to be reexecuted during replay, or executed before the checkpoint and thus should not be reexecuted during replay.

A naive solution to the above problem would be to delay the checkpointing of a thread if it is in execution anywhere between the beginning of the before hook and the end of

the after hook. However, this could delay checkpointing for arbitrarily long time if a thread is blocked on a system call, such as *read()*.

The actual solution in *RRC* has two properties: (I) checkpointing of a thread is delayed if the thread is *within* the before hook or *within* the after hook, and (II) checkpointing of a thread can occur even if the thread is between the end of the before hook and the beginning of the after hook.

To enforce property (I), each thread registers with the kernel the address of a per-thread *in_rr* variable. In user mode, the RR library sets/clears the *in_rr* when it respectively enters/leaves the hook function. An addition to the kernel code prevents the thread from being paused if the thread's *in_rr* flag is set.

To deal with property (II), *RRC* includes mechanisms to: (A) detect that this scenario has occurred, and (B) eliminate the potential ambiguities, such as the one mentioned above and take appropriate actions during replay. To implement the required mechanisms, *RRC* uses three variables: two per-thread flags – *in_hook* and *syscall_skipped*, as well as a global *current_phase* variable [63]. These variables are shared between the user level and the kernel. In the record phase, *in_hook* is set in the before hook and cleared in the after hook – this is mechanism (A) above.

For mechanism (B), *syscall_skipped* is used, during the replay phase, to determine whether, during the record phase, the checkpoint was taken before or after executing the system call. During the record phase, this flag is cleared during initialization and is not otherwise read or written. With CRIU (§2.1), if a checkpoint is triggered while a thread is executing a system call, before that call performs any state changes, the system call is retried after the checkpoint is restored. In the replay phase, at an early point in the kernel code executing a system call, if *in_hook* is set, the system call is skipped and *syscall_skipped* is set. Thus, if the system call was not executed before the checkpoint, it will be initially skipped during replay. During replay, if the after hook finds that *in_hook* and *syscall_skipped* are set, it passes control back to the before hook and the system call is then replayed or re-executed.

The handling of lock operations is similar to the handling of system calls. In the after hook, if *in_hook* is set, the lock is released and control is passed to the before hook, thus allowing enforcement of the order of lock acquires.

4.3 Handling Network Traffic

The current *RRC* implementation assumes that all network traffic is via TCP. To ensure failure transparency with respect to clients, there are three requirements that must be met: (1) client packets that have been acknowledged must not be lost; (2) packets to the clients that have not been acknowledged may need to be resent; (3) packets to the clients must not be released until the backup is able to recover the primary state past the point of sending those packets.

Requirements (1) and (2) have been handled in connection with other mechanisms, such as [20, 60]. With *RRC*, this is done by routing incoming and outgoing packets through the backup (§3). Incoming packets are recorded by the PackRec thread in the agent. Outgoing packets are sent to the backup as part of the nondeterministic event log.

The PackGate kernel module on the backup is used to meet requirement (3). PackGate maintains a release sequence number (*RSN*) for each TCP stream. When the primary container sends an outgoing message, the nondeterministic event log it sends to the backup (§3) includes a release request that updates the stream's *RSN*. The outgoing packets with sequence numbers lower than the *RSN* are then released.

PackGate is implemented in the kernel since it operates frequently and must thus be efficient. PackGate maintains fairness among the TCP streams using a FIFO queue of release requests ordered by the order of sends.

4.4 Transition to Live Execution

As with [38, 43] and unlike the deterministic replay tools for debugging [44, 55–57], *RRC* needs to transition from replay mode to live mode. This occurs when the backup replica finishes replaying the nondeterministic event log, specifically, when the last system call that generated an external output during the original execution is replayed. To identify this last call, after the checkpoint is restored, the RR library scans the nondeterministic event log and counts the number of system calls that generated an external output. Once replay starts, this count is atomically decremented and the transition to live execution is triggered when the count reaches 0.

To support live execution, after replay, the kernel state must be consistent with the state of the container and with the state of the external world. For most kernel state, this is achieved by actually executing during replay system calls that change kernel state. For example, this is done for system calls that change the file descriptor table, such as *open()*, or change the memory allocation, such as *mmap()*. However, this approach does not work for system calls that interact with the external world. Specifically, in the context of *RRC*, these are reads and writes on sockets associated with a connection to an external client. As discussed in §4.1, such calls are replayed from the nondeterministic event log. However, there is still a requirement of ensuring that, before the transition to live execution, the state of the socket, e.g., sequence numbers, must be consistent with the state of the container and with the state of external clients.

To overcome the above challenge, when replaying system calls that affect socket state, *RRC* records the state changes on the sockets based on the nondeterministic event logs. When the replay phase completes, *RRC* updates all the sockets based on the recorded state. Specifically, the relevant components of socket state are: the last sent sequence number, the last acknowledged (by the client) sequence number, the last re-

ceived (from the client) sequence number, the receive queue, and the write queue. The initial socket state is obtained from the checkpoint. The updates to the sent sequence number and the write queue contents are determined based on writes and sends in the nondeterministic event log. For the rest of the socket state, *RRC* cannot rely on the event log since some packets received and acknowledged by the kernel may not have been read by the application. Instead, *RRC* uses information obtained from PackRec (§4.3).

With respect to incoming packets, once the container transitions to live execution, *RRC* must provide to the container all the packets that were acknowledged by the primary but were not read by applications. During normal operation, on the backup host, PackRec keeps copies of incoming packets while PackGate extracts the acknowledgment numbers on each outgoing stream. If the primary fails, PackGate stops releasing outgoing packets and it thus has the last acknowledged sequence number of each incoming stream. PackRec obtains the last acknowledged sequence number of each stream from PackGate and stops recording when it has all the required (acknowledged) incoming packets. Before the container is restored on the backup, PackRec copies the recorded incoming packets to a log. Using the information from the nondeterministic event log and PackRec, before the transition to live execution, the packet repair mode (§2.1) is used to restore the socket state so that it is consistent with the state of the container and the external world.

4.5 Transferring the Event Logs

Whenever the container on the primary sends a message to an external client, it must collect the corresponding entries from the multiple nondeterministic event logs (§4.1) and send them to the backup (§3). Hence, the collection and sending of the log is a frequent activity, which is thus performance critical. Specifically, with our initial implementation, with the *Memcached* benchmark under maximum load, the throughput overhead was approximately 300%.

To address the performance challenge above, *RRC* offloads the transfer of the nondeterministic event log from the application threads to a dedicated *logging thread* added by the RR library to the application process (as in [47]). With available CPU cycles, such as additional cores, this minimizes the overhead for the application threads. Furthermore, if multiple application threads generate external messages at approximately the same time, the corresponding multiple transfers of the logs are batched together, further reducing the overhead. When an application thread sends an external message, it notifies the logging thread via a shared ring buffer. The logging thread continuously collects all the notifications in the ring buffer and then collects and sends the nondeterministic logs to the backup. To reduce CPU usage and enable more batching, the logging thread sleeps for the minimum time allowed by the kernel between scans of the buffer.

To maximize performance, *RRC* allows concurrent access to different logs. One application thread may log a lock operation concurrently with another thread that is logging a system call, while the logging thread is collecting log entries from a third log for transfer to the backup. This enables the logging thread to collect entries from different logs out of execution order. Thus, there is the potential for the log transferred to the backup for a particular outgoing message to be incomplete – missing an entry for an event on which the outgoing message depends. This can lead to replay failure.

There are two key properties of *RRC* that help address the correctness challenge above: (A) there is no need to replay the nondeterministic event log beyond the last system call that outputs to the external world, and (B) when an application thread logs a system call that outputs to the external world, all nondeterministic events on which this system call may depend are already logged in nondeterministic event logs.

To exploit the two properties above, the RR library maintains two corresponding global sequence numbers: *primary batch sequence number* (PBSN) and *backup batch sequence number* (BBSN) in the primary and backup, respectively. They are both initialized to 0. Application threads attach the PBSN to the entries they log for nondeterministic events. When the logging thread picks up an entry from the aforementioned ring buffer, that is a request to collect and send the current event log. Before taking any other action, the logging thread scans the ring buffer to determine the number of pending requests. It then increments the PBSN by that number. Thus, every event log entry that is created after the logging thread begins collecting the log, has a higher PBSN tag. After the logging thread sends the log, it sends to the backup a message that directs the backup to increment the BBSN by the most recent increment of the PBSN. If the primary fails, before replay is initiated on the backup, all the nondeterministic event logs collected during the current epoch are scanned and the entries for system calls that output to the external world are counted *if* their attached sequence number is not greater than the BBSN. During replay, this count is decremented for each such system call replayed. When it reaches 0, replay terminates and live execution commences.

4.6 Container Fork

The new container fork (*cfork*) system call is based on the existing process *fork*. Given a process ID in a container, *cfork* duplicates the container state shared among its processes and threads: namespaces (e.g., mount points, network interfaces) and control groups. *Cfork* then duplicates all the processes and their threads in the container and assigns them to the new container. *Fork* duplicates the file descriptor state, but does not duplicate the underlying state, such as socket state or pipe state. However, *cfork* does duplicate this underlying state.

The implementation of *cfork* for *RRC* includes optimizations to minimize the container fork time. We identified two

major sources of overhead: (1) duplicating the namespaces and control groups, and (2) page table copy. To minimize (1), *RRC* exploits the fact that most namespace and control group state rarely changes after initialization [62]. Thus, at the first checkpoint, *RRC* creates a staging container with an idle process. *Cfork* assigns the forked container to the namespace and control group of the staging container instead of creating new ones. To ensure correctness, *RRC* detects state changes of the namespaces and control groups of the service container using hooks, added with *ftrace* to the kernel functions that can change the namespace and cgroup state. *Cfork* updates those changes to the staging container. *Ftrace* only incurs overhead if a hooked function is invoked. Since the namespace and cgroup states rarely change, such functions are rarely invoked and *ftrace* does not incur high overhead.

To minimize the latency of the page table copy, *RRC* avoids copying the page table of the data region of the RR library, whose size can be up to several gigabytes and thus takes tens of milliseconds to copy. Specifically, the RR library tags the VMA of the data region with a new special flag and thus informs the *cfork* to skip copying its page table. This optimization is correct because *RRC* does not need to checkpoint the data region of the RR library; its state is initialized upon replay by reading the saved nondeterministic logs in the backup.

4.7 Mitigating the Impact of Data Races

As discussed in §3, *RRC* includes mechanisms that significantly increase the probability of successful recovery in the presence of rarely-manifested data races. Specifically, *RRC* mitigates the impact of data races by adjusting the relative timing of the application threads during replay to approximately match the timing during the original execution. As a first step, in the record phase, the RR library records the order and the TSC (time stamp counter) value when a thread leaves the after hook of a system call. In the replay phase, the RR library enforces the recorded order on threads before they leave the after hook. As a second step, during replay, the RR library maintains the TSC value corresponding to the time when the after hook of the last-replayed system call was exited. When a thread is about to leave a system call after hook, the RR library delays the thread until the difference between the current TSC and the TSC of that last-replayed system call is larger than the corresponding difference in the original execution. System calls are used as the basis for the timing adjustments since they are replayed (not executed) and are thus likely to cause the timing difference. This mechanism is evaluated in §6.3.

5 Experimental Setup

All the experiments were hosted on Fedora 29 with the 4.18.16 Linux kernel. The containers were hosted using runC [12] (version 1.0.1), a popular container runtime used in Docker. The primary and backup replicas were hosted on different

36-core servers, using modern Xeon chips. These hosts were connected to each other through a dedicated 10Gb Ethernet link. The clients were hosted on a 10-core server, based on a similar Xeon chip. The client host was in a different building, interconnected through a Cisco switch, using 1Gb Ethernet.

Five benchmarks were in-memory databases handling short requests: *Redis* [13], *Memcached* [10], *SSDB* [15], *Tarantool* [16] and *Aerospike* [2]. These benchmarks were evaluated with 50% read and 50% write requests to 100,000 100B records, driven by *YCSB* [31] clients. The number of user client threads ranged from 60 to 480. The evaluation also included a web server, *Lighttpd* [7], and two batch PARSEC [26] benchmarks: *Swaptions* and *Streamcluster*. *Lighttpd* was evaluated using 20-40 clients retrieving a 1KB static page. For *Lighttpd*, benchmarking tools SIEGE [14], ab [1] and wget [5] were used to evaluate, respectively, the performance overhead, response latency, and recovery rate. *Swaptions* and *Streamcluster* were evaluated using the native input test suites. We evaluated only two benchmarks from the PARSEC suite since *RRC* targets server applications and its design is thus focused on low latency overhead. Low latency overhead is not relevant for the batch applications, such as those included in the PARSEC suite. Nonetheless, we show that such applications can be handled by *RRC* with very low throughput overhead.

We used fault injection to evaluate *RRC*'s recovery mechanism. Since fail-stop failures are assumed, a simple failure detector was sufficient. Failures were detected based on heart beats exchanged every 30ms between the primary and backup hosts. The side not receiving heart beats for 90ms identified the failure of the other side and initiates recovery.

For *Swaptions* and *Streamcluster*, recovery was “successful” if the output was identical to the golden copy. For *Lighttpd*, we used multiple wget instances that concurrently fetched a static page. Recovery was “successful” if all the fetched pages were identical to the golden copy. For the in-memory database benchmarks, we developed customized clients, using existing client libraries [3, 9, 11, 17], that spawn multiple threads and let each thread work on separate set of database records. Each thread records the value it stores with each key, compares that value with the value returned by the corresponding get operation and flags an error if there is a mismatch. Recovery was considered successful if no errors were reported.

For the fault injection experiments, for server programs, the clients were configured to run for at least 30 seconds and drive the server program to consume around 50% of the CPU cycles. A fail stop failure was injected at a random time within the middle 80% of the execution time, using the *sch_plug* module to block network traffic on all the interfaces of a host. To emulate a real world cloud computing environments, while also stressing the recovery mechanism, we used a *perturb* program to compete for CPU resources on the primary host. The *perturb* program busy loops for a random time between 20 to 80 ms and sleeps for a random time between 20 to 120ms. During fault injection, a *perturb* program instance

	TP Overhead			Avg. Latency(μ s)		
	Redis	Taran	Aero	Redis	Taran	Aero
Custom	49%	31%	153%	574	471	456
<i>RRC-LE</i>	31%	26%	47%	543	564	602

Table 1: Throughput overhead and average latency. *RRC* vs. custom replication mechanisms.

was pinned to each core executing the benchmark.

6 Evaluation

This section presents *RRC*'s performance overhead and CPU usage overhead (§6.1), the added latency for server responses (§6.2), as well as the recovery rate and recovery latency (§6.3). Two configurations of *RRC* are evaluated: *RRC-SE* (short epoch) and *RRC-LE* (long epoch), with epoch durations of 100ms and 1s, respectively. Setting the epoch duration is a tradeoff between the lower overhead with long epochs and the lower susceptibility to data races and lower recovery time with short epochs. Hence, *RRC-LE* may be used if there is high confidence that the applications are free of data races. Thus, with the *RRC-SE* configuration, the data race mitigation mechanism described in §4.7 is turned on, while it is turned off for *RRC-LE*.

RRC is compared to NiLiCon (§2.1) with respect to the performance overhead under maximum CPU utilization and the server response latency. NiLiCon is configured to run with an epoch interval of 30ms, as in [62]. The short epochs of NiLiCon are required since, unlike *RRC*, the epoch duration with NiLiCon determines the added latency in replying to client requests (§2.1). Thus, for many server applications, even with 30ms epochs, NiLiCon provides unacceptably long response latencies. In all cases, the “stock setup” is the application running in an unreplicated container.

Some server applications can be configured to enable their own custom fault tolerance mechanisms. However, developing and validating such mechanisms is time consuming and error prone. Hence, mechanisms, such as *RRC*, that can be deployed for many applications, are likely to be of higher quality (reliability) and incur lower total development cost. Table 1 compares the overhead of *RRC* with the custom mechanisms of three of our benchmarks (§5). The custom mechanisms are all configured to provide *strong consistency* (outputs are not released until the changes are reflected in the backup), which *RRC* also provides. The results show that *RRC-LE* actually has lower throughput overhead. On average, the custom mechanisms do result in lower response latency. This is mainly due to their ability to release the outputs of read requests without waiting for acknowledgments from the backup. However, on average, the overall results are comparable.

6.1 Overheads: Performance, CPU Utilization

Two key overhead measures of *RRC* are: for a fixed amount of work, the increase in execution time and the increase in the utilization of CPU cycles. These measures are distinct

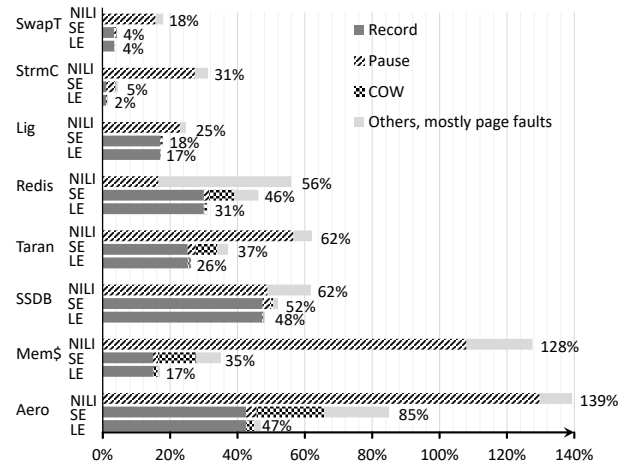


Figure 3: Performance overheads: NiLiCon, *RRC-SE*, *RRC-LE*.

since many of the actions of *RRC* are in parallel with the main computation threads.

For the six server benchmarks, the measurements reported in this subsection were done with workloads that resulted in maximum CPU utilization for the cores running the application worker threads¹ with the stock setup.

With four of the server benchmarks, the number of the worker threads cannot be configured (*Lighttpd*, *Redis*: 1, *Tarantool*: 2, *SSDB*: 12). The remaining four benchmarks were configured to run with four worker threads.

For each benchmark, the workload that saturates the cores in the stock setup was used for the stock, *RRC*, and NiLiCon setups. With NiLiCon, due to its large latency overhead (§6.2), it is impossible to saturate the server with this setup. Hence, for the NiLiCon measurements in this subsection, the buffering of the server responses was removed. This is not a valid NiLiCon configuration, but it provides a comparison of the overheads excluding buffering of external outputs.

Performance overhead. The performance overhead is reported as the percentage increase in the execution time for a fixed amount of work compared to the stock setup. Figure 3 shows the performance overheads of NiLiCon, *RRC-SE*, and *RRC-LE*, with the breakdown of the sources of overhead. Each benchmark was executed 50 times. The margin of error of the 95% confidence interval was less than 2%.

The record overhead is caused by the RR library recording nondeterministic events. The pause overhead is due to the time the container is paused during the container fork. The COW overhead is caused by the time to copy the pages after the container fork. The page fault overhead is caused by the page fault exceptions that track the memory state changes of each epoch (§2.1).

With *RRC-SE*, the average incremental checkpoint size per epoch was 0.2MB for *Swaptions*, 15.6MB for *Redis*, and 41.2MB for *Aerospike*. With *RRC-SE*, the average number of

¹Some application “helper threads” are mostly blocked sleeping.

WWSS	Redis	Taran	SSDB	Mem\$
1x	47% (1.00)	37% (1.00)	53% (1.00)	36% (1.00)
2x	56% (1.19)	51% (1.38)	57% (1.08)	58% (1.61)
3x	73% (1.55)	63% (1.70)	62% (1.17)	73% (2.03)

Table 2: The impact of the write working set size (WWSS), relative to the WWSS used in Figure 3, on the performance overhead with *RRC-SE*. The overheads relative to the 1x case are in parentheses.

	ST	SC	Lig	Redis	Taran	SSDB	Mem\$	Aero	
CP	Primary	3%	5%	8%	30%	16%	8%	13%	31%
	Backup	1%	2%	5%	26%	15%	4%	13%	18%
RR	Primary	4%	1%	34%	47%	46%	55%	36%	77%
	Backup	~0	~0	33%	29%	20%	11%	15%	19%
TCP	Primary	~0	~0	~0	~0	~0	~0	~0	~0
	Backup	~0	~0	59%	86%	54%	23%	40%	43%
total	8%	8%	139%	218%	151%	101%	117%	188%	

Table 3: CPU utilization overhead for *RRC-SE*. **CP**: checkpointing. **RR**: recording nondeterministic events. **TCP**: handling TCP failover.

logged lock operations plus system calls per epoch was 9 with *Streamcluster*, 907 with *Tarantool*, and 2137 with *Aerospike*, partially explaining the differences in record overhead. However, the overhead of logging system calls is much higher than for lock operations. *Memcached* is comparable to *Aerospike* in terms of the rate of logged system calls plus lock operations, but has 341 compared to 881 logged system calls per epoch and thus lower record overhead.

The number of pages modified during an epoch determines the rate of page faults and COW operations, as well as the size of the incremental checkpoint that is transferred to the backup (§3). Hence, this write working set size (WWSS) impacts the performance overhead. Table 2 shows the performance overhead of *RRC-SE* with four of our benchmarks as the WWSS is increased to 2x and 3x of the WWSS used in Figure 3. These measurements were obtained by increasing the number of records and then, with a fixed number of records, varying the ratio of writes to reads to obtain the different WWSS values. As expected, the checkpointing component of the overhead (“COW” plus “Others” in Figure 3) increases approximately linearly with the WWSS. As shown in Figure 3, as the epoch duration is increased, the checkpointing component of the overhead is decreased and thus the impact of the WWSS becomes less significant. It should be noted that the number of pages read during an epoch has no impact on the performance overhead. The footprint of the application has only negligible impact that is due to the time to scan the page table to identify the modified pages.

CPU utilization overhead. The CPU utilization (Table 3) is the product of the average numbers of CPUs (cores) used and the total execution time. The CPU utilization overhead is the percentage increase in utilization with *RRC* compared to with the stock setup. The breakdown of the overhead into its components was obtained by incrementally enabling each component and measuring the corresponding increase in CPU

	Lig1K	Lig100K	Redis	Taran	SSDB	Mem\$	Aero	
S	avg	549	2059	406	393	388	643	373
	99%	<1ms	<3ms	734	617	622	2982	711
R	avg	694	2203	604	604	651	812	663
	99%	<1ms	<3ms	969	992	988	3941	1273
N	avg	38ms	38ms	42ms	42ms	45ms	45ms	51ms
	99%	<39ms	<39ms	44ms	42ms	47ms	53ms	63ms

Table 4: Response latency in μ s. S: Stock, R: *RRC-SE*, N: NiLiCon

overhead. A significant factor in the CPU utilization overhead is for packet handling in the backup kernel needed to support TCP failover. This overhead is mostly due to routing. Techniques for optimizing software routing [42] can be used to reduce this overhead.

The overheads shown in Table 3 should be evaluated in the context of comparable alternative techniques. The only alternatives that can achieve low latency overheads necessary for many server applications are based on active replication [38, 47]. Such techniques have CPU overheads comparable to *RRC*’s for recording nondeterministic events and handling TCP failover. They do not have the overhead for checkpointing but instead have 100% overhead for execution on the backup. Table 3 shows the with *RRC-SE* the overhead for checkpointing is 4%-56%. Hence, *RRC*’s CPU overhead is significantly smaller than the comparable alternatives’.

Performance decoupling. An important property of *RRC* is that, unlike active replication, it decouples the performance of the application on the primary host from the performance on the backup. To illustrate the impact of this, we selected two representative benchmarks: *Redis* and *Aerospike*, which incur a significant CPU usage on the backup host, and ran them with *RRC-SE*. We ran the perturb program (§5), which consumes 40% of a CPU, first on all the cores of the primary and then the backup. When the perturb program runs on the primary, the performance overhead increases from 46% to 71% and 85% to 116% for *Redis* and *Aerospike*, respectively. However, when the perturb program runs on the backup, the execution time remains the same.

6.2 Response Latency

Table 4 shows the response latencies with the stock setup, *RRC-SE* and NiLiCon. The numbers of client threads for stock and *RRC-SE* are adjusted so that the CPU load on the cores running application worker threads is 50%. For NiLiCon, the number of client threads is the same as with *RRC-SE*, resulting in CPU utilization of less than 5%, thus favoring NiLiCon. To evaluate the impact of response size, *Lighttpd* is evaluated serving both 1KB as well as 100KB files.

With *RRC*, there are three potential sources for the increase in response latency: forwarding packets through the backup, the need to delay packet release until the corresponding event log is received by the backup, and increased request processing time on the primary. With *RRC-SE*, the increase

	ST	SC	Lhttpd	Redis	Taran	SSDB	Mem\$	Aero	
CF	avg	0.7	2.7	0.5	1.6	2.4	2.6	1.5	3.2
	90%	0.7	3.1	0.6	1.9	2.7	2.9	1.7	3.5
NCF	avg	5.9	7.6	7.2	14.9	18.4	13.9	28.7	42.9
	90%	5.9	8.0	7.4	16.7	20.2	14.8	33.7	45.8

Table 5: The pause time of *RRC* with container fork (CF) and without container fork (NCF) in millisecond.

	ST	SC	Lhttpd	Redis	Taran	SSDB	Mem\$	Aero
avg	3.1	3.9	2.5	9.1	11.5	6.5	15.6	27.4

Table 6: The average time (ms) between resuming container execution and the stop of COW.

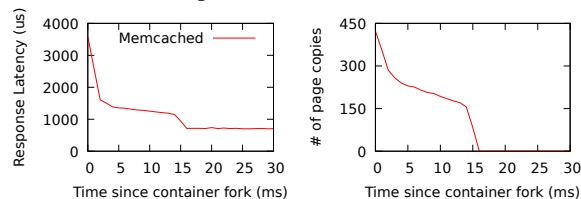


Figure 4: Average response latency and the number of COW since container fork.

in average latency is only $144\mu s$ to $290\mu s$. The worst case is with *Aerospike*, which has the highest checkpointing overhead (COW+Others in Figure 3) and a high rate of nondeterministic events and thus long logs that have to be transferred to the backup. The increase in 99th percentile latency is $235\mu s$ to $959\mu s$. The worst case is with *Memcached*. As shown in Table 4, in terms of increase in response latency, NiLiCon is not competitive, as also indicated by the results in [62].

With *RRC-LE*, the increase in the average response latency is from $42\mu s$ to only $229\mu s$, due to the the lower checkpointing overhead. The increase in the 99th percentile latency is under $510\mu s$ since the container fork are much less frequent and thus less likely to interrupt the processing of a request.

The impact of container fork. The tail response time latency overhead is determined by the time the primary is paused for checkpointing. Table 5 shows *RRC*'s pause time with and without the container fork. Without the container fork, the container has to be paused during the entire checkpointing process, leading to a pause time between 5.9ms to 45.8ms. The pause time with the container fork is only from 0.5ms to 3.5ms. Most of the container fork time is spent on copying page tables and thus can be further reduced with recent techniques on optimizing *fork()* [61].

Due to the reduction in the pause time, with the SE setup, the container fork reduces the average response latency overhead from $156\mu s$ - $581\mu s$ to $144\mu s$ - $290\mu s$, and the worst-case 99% response latency overhead from 6ms to $959\mu s$. The throughput overhead is reduced from 8%-145% to 4%-85%.

Immediately after the container fork there is a period during which there is additional overhead due to COW of pages on the primary (§3). Table 6 shows that this period terminates at an early stage of each epoch. To evaluate the impact of the COW on response latency, we obtained fine grained measurements with *Memcached*. Figure 4 shows the results.

		Recovery Rate		Replay Time	
		Mem\$	Aero	Mem\$	Aero
100ms	stock	94.3%	84.5%	20	28
	+ Total order of syscalls	94.3%	92.7%	131	299
	+ Timing adjustment	99.2%	99.8%	234	383
1s	stock	51.4%	34.8%	249	373
	+ Total order of syscalls	51.6%	76.5%	1122	1345
	+ Timing adjustment	99.0%	99.4%	1230	1460

Table 7: Recovery rate and replay time (in ms). *RRC* with different levels of mitigation of data race impact.

Immediately after the container fork, due to the pause and a high rate of page copies, the response latency is around 3.5ms. However, the response latency almost immediately drops to around 1.5ms and then to $700\mu s$, where it remains for the rest of the epoch.

6.3 Recovery Rate and Latency

This subsection presents an evaluation of the recovery mechanism and the data race mitigation mechanism. The service interruption time is obtained by measuring, at the client, the increase in response latency when a fault occurs. The service interruption time is the sum of the recovery latency plus the detection time. With *RRC*, the average detection time is 90ms (§5). Hence, since our focus is not on detection mechanisms, the average recovery latency reported is the average service interruption time minus 90ms.

Backup failure. 50 fault injection runs are performed for each benchmark. Recovery is always successful. The service interruption duration is dominated by the Linux TCP retransmission timeout, which is 200ms. The other recovery events, such as detector timeout and broadcasting the ARP requests to update the service IP address, occur concurrently with this 200ms. Thus, the measured service interruption duration is between 203ms and 208ms.

Primary failure recovery rate. Three of our benchmarks contain data races that may cause recovery failure: *Memcached*, *Aerospike*, and *Tarantool*. Running *Tarantool* with *RRC-SE*, through 50 runs of fault injection in the primary, we find that, due to data races, in all cases replay fails and thus recovery fails. Due to the high rate of data race manifestation, this is the case even with the mechanism described in §4.7. Thus, we use a version of *Tarantool* in which the data races are eliminated by manually adding locks.

We divide the benchmarks into two sets. The first set consists of the five data-race-free benchmarks and a modified version of *Tarantool*. For these, 50 fault injections are performed for each benchmark. Recovery is always successful.

The second set of benchmarks, *Memcached* and *Aerospike*, is used to evaluate the the data race mitigation mechanisms (§4.7). For these, to ensure statistically significant results, 1000 fault injection runs are performed with each benchmark with each setup. The results are presented in Table 7. For both the recovery rate and replay time, the 95% confidence interval

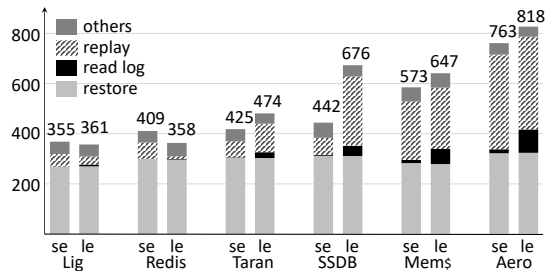


Figure 5: Recovery latency (ms) breakdown with *RRC-SE* and *RRC-LE*.

is less than 1%. Without the §4.7 mechanism, the recovery rate for *RRC-LE* is much lower than with *RRC-SE*, demonstrating the benefit of short epochs and thus shorter replay times. Enforcing a total order of the recorded system calls in the after hook is not effective for *Memcached* but increases the recovery rate of *Aerospike* for both *RRC* setups. However, with the timing adjustments, both benchmarks achieve high recovery rates, even with *RRC-LE*. The total order of the system calls is the main factor that increase the replay time. Thus, there is no reason to not also enable the timing adjustments.

We measured the rate of racy memory accesses in *Tarantool*, *Memcached* and *Aerospike*. To identify “racy memory accesses”, we first fixed all the identified data races by protecting certain memory access with locks. We then removed the added locks and added instrumentation to count the corresponding memory accesses. For *Tarantool*, the rates of racy memory writes and reads are, respectively, 328,000 and 274,000 per second. For *Memcached* the respective rates are 1 and 131,000 per second and for *Aerospike* they are 250 and 372,000 per second. These results demonstrate that, when the rate of accesses potentially affected by data races is high, our mitigation scheme is not effective. Fortunately, in such cases, data races are unlikely to remain undetected.

Primary failure recovery latency. Figure 5 shows a breakdown of the factors that make up the recovery latency for the server benchmarks with *RRC-SE* and *RRC-LE*. With *RRC-SE*, the data race mitigation scheme is enabled, while with *RRC-LE* it is disabled. The 95% confidence interval margin of error is less than 5%. *Restore* is the time to restore the checkpoint, mostly for restoring the in-kernel states of the container (e.g., mount points and namespaces). *Read log* is the time to process the stored logs in preparation for replay. *Others* include the time to send ARP requests and connect the backup container network interface to the bridge.

The recovery latency differences among the benchmarks are due mainly to the replay time. It might be expected that the average replay time would be approximately half an epoch duration. However, replay time is increased due to different thread scheduling by the kernel that causes some threads to wait to match the order of the original execution. This increase is more likely when the data race impact mitigation mechanism is enabled since it enforces more strict adherence

footprint	Redis	Taran	SSDB	Mem\$	Aero
1x	409 (1.00)	425 (1.00)	442 (1.00)	573 (1.00)	763 (1.00)
2x	424 (1.04)	460 (1.08)	479 (1.08)	583 (1.02)	836 (1.10)
3x	463 (1.13)	493 (1.16)	524 (1.18)	609 (1.06)	917 (1.20)

Table 8: The impact of the footprint size, relative to the footprint size used in Figure 5, on the primary recovery latency (in ms) with *RRC-SE*. The latencies relative to the 1x case are in parentheses.

to the original execution. A second factor that impact the replay time is a decrease due to system calls that are replayed from the log and not executed.

With the current *RRC* implementation, the total memory occupancy of the application, i.e., its footprint, has an impact on the recovery latency. Specifically, during recovery on the backup host, all the pages are copied from the memory area where they are saved during prior checkpointing to new locations. Hence, as shown in Table 8, as the footprint is increased, there is a small increase in the recovery latency. In these measurements, the footprint was determined by the final checkpoint size. It should be noted that the impact of the footprint on recovery latency is a limitation of the current implementation. An optimization with kernel support would avoid copying the pages from one memory location to another by simply updating the page table.

7 Limitations

An inherent limitation is that the mechanism used for mitigating the impact of data races (§4.7) is incapable of handling a high rate of racy accesses (§6.3). However, as discussed in §3, such data races are easily detectable and are thus easy to eliminate, even in legacy applications.

The prototype implementation of *RRC* is restricted to single-process containers. This is not a major restriction since, in most cases, containers are used to run only a single process. Cito et al. [30] analyzed 38,079 Docker projects on Github and concluded that only 4% of the projects involved multi-process containers. This is reinforced by Internet searches regarding this issue that yield numerous hits on pages, such as [23], that suggest that running single-process containers is best practice. To overcome this limitation, the *RR* library would need to support inter-process communications via shared memory. Techniques presented in [24] may be applicable.

RRC also does not handle asynchronous signals. This can be resolved by techniques used in [43], that delay signal delivery until a system call or certain page faults.

The current implementation of *RRC* only supports C/C++ applications. Adding support for non-C/C++ applications would require instrumenting their runtimes to track nondeterministic events. *RRC* does not handle C atomic types, functions, intrinsics and inline assembly code that performs atomic operations transparently. In this work, such cases were handled by protecting such operations with locks.

8 Related Work

RRC is related to prior fault-tolerance works on replication based on high-frequency checkpointing, replication based on deterministic replay, and network connection failover.

Early work on VM replication is based on leader-follower active replication using deterministic replay [27]. This is combined with periodic checkpointing in [28], based on use of this technique for debugging [41]. These works focused on uniprocessor systems. Extending them to multiprocessors is impractical, due to the overhead of recording shared memory access order for a VM [37, 52]. Remus [33] (§2.1) and its follow-on works [46, 53, 62] focus on multiprocessor workloads and implement replication using high-frequency checkpointing. Plover [58] optimizes Remus by using an active replica to reduce the size of transferred state and by performing state synchronization adaptively, when VMs are idle. All the Remus-based mechanisms release outputs only after the primary and backup synchronize their states. Hence, outputs are delayed by multiple (often, tens of) milliseconds. COLO [35] compares outputs from two active VM replicas and synchronizes their states on a mismatch, resulting in high throughput and latency overheads for applications with significant nondeterminism.

For process-level checkpointing, libckpt [51] implements “forked checkpointing,” where the unmodified *fork()* system call is used to minimize the pause time for checkpointing.

To handle nondeterminism in parallel applications, as with *RRC*, some works rely on replaying the order of synchronization operations [32, 38, 47]. Rex [38] and Crane [32] cannot handle state divergences caused by data races and require manual modifications of the application source code. Castor [47] handles data races by buffering outputs until the backup finishes replaying the associated logs. If divergence due to data races occurs, the two replicas synchronize their state.

Comparing *RRC* with Rex, Crane, and Castor, for data-race-free applications, *RRC* is likely to have a smaller throughput overhead. Specifically, Rex reports that under heavy load, replay may be slower than the original execution and thus the active replica is a performance bottleneck. With a data-race-free setup, both Rex and *RRC* are evaluated with *Memcached*, and the performance overheads are 40% vs. 17%.

For applications that have data races, the only relevant comparison is with Castor. Castor is likely to have higher response delays since outputs cannot be released until the backup finishes replaying the associated log. Additionally, a data race can also cause Castor to fail. Specifically, if the primary fails in the middle of state synchronization caused by a data race, the system fails. Hence, for an application with a high rate of racy memory accesses, such as *Tarantool* (§6.3), Castor would be frequently synchronizing the state and thus have low recovery rate (like *RRC*) and also high performance overhead. For applications with a lower rate of racy memory accesses, such as *Memcached* and *Aerospike*, Castor also has lower recovery rate. For example, for *Memcached*, based on Table 7, the probability of execution divergence in 50ms is 0.059. Hence,

execution diverges approximately every 0.85s. With our setup, the time it takes to create and transfer the checkpoint for *Memcached* is 48ms. Hence, an upper bound on the recovery rate with Castor is expected to be 94.7% versus 99.2% with *RRC* (Table 7). A similar calculation for *Aerospike*, taking into account 76ms to create and transfer the checkpoint, results in a recovery rate for Castor of 79.8% versus 99.8% for *RRC*.

9 Conclusion

RRC is a unique point in the design space of application-transparent fault tolerance schemes for multiprocessor workloads. By combining checkpointing, with externally deterministic replay, and container fork, it provides all the desirable properties of a fault tolerance scheme listed in §1, with specific emphasis on low latency overhead, which is critical for server applications. *RRC* facilitates trading off performance and resource overheads with vulnerability to data races and recovery latency. Critically, the response latency is decoupled from the frequency of checkpointing, and sub-millisecond added delay is achieved with all our server applications. *RRC* is a full fault tolerance mechanism. It can recover from primary or backup host failure and includes transparent failover of TCP connections.

As we have found (§6.3), legacy applications may have data races. *RRC* targets data races that are most likely to remain undetected and uncorrected, namely, rarely-manifested data races. Unlike mechanism based strictly on active replication and deterministic replay [38], *RRC* is not affected by data races that manifest during normal operation, long before failure. For data races that manifest right before failure, *RRC* introduces simple mechanisms that significantly reduce the probability of the data races causing recovery failure.

This paper describes key implementation challenges encountered in the development of *RRC* and outlines their resolution. The extensive evaluation of *RRC*, based on eight benchmarks, included performance and resource overheads, impact on response latency, as well as recovery rate and latency. The recovery rate evaluation, based on fault injection, subjected *RRC* to particularly harsh conditions by intentionally perturbing the scheduling on the primary, thus challenging the deterministic replay mechanism (§5). With high checkpointing frequency (*RRC-SE*), *RRC*’s throughput overhead is less than 53% for seven of our benchmarks and 85% for the eighth. If the applications are known to be data-race-free, with a lower checkpointing frequency (*RRC-LE*), the overhead is less than 49% for all benchmarks, significantly outperforming NiLiCon [62]. With data-race-free applications, *RRC* recovers from all fail-stop failures. With two applications with infrequently-manifested data races, the recovery rate is over 99% with *RRC-SE*.

Acknowledgments

We thank our reviewers, especially our shepherd, for constructive feedback that significantly improved this paper.

References

- [1] ab - Apache HTTP server benchmarking tool. <https://httpd.apache.org/docs/2.4/programs/ab.html>.
- [2] Aerospike. <https://www.aerospike.com/>.
- [3] Aerospike C Client. <https://www.aerospike.com/apidocs/c/>.
- [4] CRIU: Checkpoint/Restore In Userspace. https://criu.org/Main_Page.
- [5] GNU Wget. <https://www.gnu.org/software/wget/>.
- [6] Google threadsanitizer. <https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual>.
- [7] Home - Lighttpd. <https://www.lighttpd.net/>.
- [8] Install Xen 4.2.1 with Remus and DRBD on Ubuntu 12.10. https://wiki.xenproject.org/wiki/Install_Xen_4.2.1_with_Remus_and_DRBD_on_Ubuntu_12.10.
- [9] libMemcached. <https://libmemcached.org/libMemcached.html>.
- [10] memcached. <https://memcached.org>.
- [11] Minimalistic C client for Redis. <https://github.com/redis/hiredis>.
- [12] opencontainers/runc. <https://github.com/opencontainers/runc>.
- [13] Redis. <https://redis.io>.
- [14] Siege Home. <https://www.joedog.org/siege-home/>.
- [15] SSDB - A fast NoSQL database, an alternative to Redis. <https://github.com/ideawu/ssdb>.
- [16] Tarantool - In-memory DataBase. <https://tarantool.io>.
- [17] Tarantool C client libraries. <https://github.com/tarantool/tarantool-c>.
- [18] TCP connection repair. <https://lwn.net/Articles/495304/>.
- [19] Navid Aghdaie and Yuval Tamir. Client-Transparent Fault-Tolerant Web Service. In *20th IEEE International Performance, Computing, and Communications Conference*, pages 209–216, Phoenix, AZ, April 2001.
- [20] Navid Aghdaie and Yuval Tamir. CoRAL: A Transparent Fault-Tolerant Web Service. *Journal of Systems and Software*, 82(1):131–143, January 2009.
- [21] Gautam Altekar and Ion Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *ACM SIGOPS 22nd Symposium on Operating Systems Principles*, page 193–206, Big Sky, Montana, USA, October 2009.
- [22] Lorenzo Alvisi, Thomas C. Bressoud, Ayman El-Khashab, Keith Marzullo, and Dmitrii Zagorodnov. Wrapping Server-Side TCP to Mask Connection Failures. In *IEEE INFOCOM*, pages 329–337, Anchorage, AK, April 2001.
- [23] Rafael Benevides. 10 Things to Avoid in Docker Containers. <https://developers.redhat.com/blog/2016/02/24/10-things-to-avoid-in-docker-containers>, February 2016. Accessed: 2022-04-25.
- [24] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic Process Groups in dOS. In *9th USENIX Conference on Operating Systems Design and Implementation*, page 177–191, Vancouver, BC, Canada, October 2010.
- [25] David Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, September 2014.
- [26] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.
- [27] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based Fault Tolerance. In *15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, USA, December 1995.
- [28] Peter M. Chen, Daniel J. Scales, Min Xu, and Matthew D. Ginzton. Low Overhead Fault Tolerance Through Hybrid Checkpointing and Replay, August 2016. Patent No. 9,417,965 B2.
- [29] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. Deterministic Replay: A Survey. *ACM Computing Surveys*, 48(2):17:1–17:47, September 2015.
- [30] Jurgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. An Empirical Analysis of the Docker Container Ecosystem on GitHub. In *IEEE/ACM 14th International Conference on Mining Software Repositories*, pages 323–333, Buenos Aires, Argentina, May 2017.

- [31] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *1st ACM Symposium on Cloud Computing*, pages 143–154, June 2010.
- [32] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos Made Transparent. In *25th Symposium on Operating Systems Principles*, pages 105–120, Monterey, California, October 2015.
- [33] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, April 2008.
- [34] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM*, 34(1):77–97, January 1987.
- [35] YaoZu Dong, Wei Ye, YunHong Jiang, Ian Pratt, ShiQing Ma, Jian Li, and HaiBing Guan. COLO: COarse-grained LOfk-stepping Virtual Machines for Non-stop Service. In *4th ACM Annual Symposium on Cloud Computing*, Santa Clara, CA, October 2013.
- [36] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *5th Symposium on Operating Systems Design and Implementation*, pages 211–224, Boston, MA, USA, December 2003.
- [37] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, page 121–130, Seattle, WA, USA, March 2008.
- [38] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the Speed of Multi-Core. In *9th European Conference on Computer Systems*, pages 161–174, Amsterdam, The Netherlands, April 2014.
- [39] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An Application-Level Kernel for Record and Replay. In *8th USENIX Conference on Operating Systems Design and Implementation*, page 193–208, San Diego, CA, USA, December 2008.
- [40] Derek R. Hower and Mark D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *35th Annual International Symposium on Computer Architecture*, page 265–276, Beijing, China, June 2008.
- [41] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *2005 USENIX Annual Technical Conference*, pages 1–15, Anaheim, CA, USA, April 2005.
- [42] Eddie Kohler, Robert Morris, Benjie Chen, and John Jannotti and Frans M. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.
- [43] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, page 155–166, New York, New York, USA, June 2010.
- [44] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 77–90, Pittsburgh, Pennsylvania, USA, March 2010.
- [45] Wubin Li, Ali Kanso, and Abdelouahed Gherbi. Leveraging Linux Containers to Achieve High Availability for Cloud Services. In *IEEE International Conference on Cloud Engineering*, pages 76–83, March 2015.
- [46] Jacob R. Lorch, Andrew Baumann, Lisa Vlendenning, Dutch Meyer, and Andrew Warfield. Tardigrade: Leveraging Lightweight Virtual Machines to Easily and Efficiently Construct Fault-Tolerant Services. In *12th USENIX Symposium on Networked Systems Design and Implementation*, Oakland, CA, May 2015.
- [47] Ali Jose Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards Practical Default-On Multi-Core Record/Replay. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, page 693–708, Xi’an, China, April 2017.
- [48] Robert H. B. Netzer and Barton P. Miller. What Are Race Conditions?: Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.
- [49] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient Deterministic Multithreading in

- Software. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, page 97–108, Washington, DC, USA, March 2009.
- [50] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09*, page 177–192, Big Sky, Montana, USA, October 2009.
- [51] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. In *USENIX 1995 Technical Conference*, pages 213–224, New Orleans, LA, January 1995.
- [52] Shiru Ren, Le Tan, Chunqi Li, Zhen Xiao, and Weijia Song. Samsara: Efficient Deterministic Replay in Multiprocessor Environments with Hardware Virtualization Extensions. In *2016 USENIX Conference on Usenix Annual Technical Conference*, page 551–564, Denver, CO, USA, June 2016.
- [53] Shiru Ren, Yunqi Zhang, Lichen Pan, and Zhen Xiao. Phantasy: Low-Latency Virtualization-based Fault Tolerance via Asynchronous Prefetching. *IEEE Transactions on Computers*, 68(2):225–238, February 2019.
- [54] Michiel Ronsse and Koen De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.
- [55] Yasushi Saito. Jockey: A User-Space Library for Record-Replay Debugging. In *Sixth International Symposium on Automated Analysis-Driven Debugging*, page 69–76, Monterey, California, USA, September 2005.
- [56] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *2004 USENIX Annual Technical Conference*, pages 29–44, Boston, MA, USA, June 2004.
- [57] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 15–26, Newport Beach, California, USA, March 2011.
- [58] Cheng Wang, Xusheng Chen, Weiwei Jia, Boxuan Li, Haoran Qiu, Shixiong Zhao, and Heming Cui. PLOVER: Fast, Multi-core Scalable Virtual Machine Fault-tolerance. In *15th USENIX Symposium on Networked Systems Design and Implementation*, pages 483–499, Renton, WA, April 2018.
- [59] Min Xu, Rastislav Bodik, and Mark D. Hill. A “Flight Data Recorder” for Enabling Full-System Multiprocessor Deterministic Replay. In *30th Annual International Symposium on Computer Architecture*, page 122–135, San Diego, California, USA, May 2003.
- [60] Dmitrii Zagorodnov, Keith Marzullo, Lorenzo Alvisi, and Thomas C. Bressoud. Practical and Low-Overhead Masking of Failures of TCP-Based Servers. *ACM Transactions on Computer Systems*, 27(2):4:1–4:39, May 2009.
- [61] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. On-demand-fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications. In *16th European Conference on Computer Systems*, pages 540–555, Virtual, April 2021.
- [62] Diyu Zhou and Yuval Tamir. Fault-Tolerant Containers Using NiLiCon. In *34th IEEE International Parallel and Distributed Processing Symposium*, pages 1082–1091, New Orleans, LA, May 2020.
- [63] Diyu Zhou and Yuval Tamir. HyCoR: Fault-Tolerant Replicated Containers Based on Checkpoint and Replay. *Computing Research Repository*, arXiv:2101.09584 [cs.DC], January 2021.



uKharon: A Membership Service for Microsecond Applications

Rachid Guerraoui¹, Antoine Murat¹, Javier Picorel², Athanasios Xygkis¹, Huabing Yan², and Pengfei Zuo²

¹*École Polytechnique Fédérale de Lausanne (EPFL)*

²*Huawei Technologies*

Abstract

Modern data center fabrics open the possibility of microsecond distributed applications, such as data stores and message queues. A challenging aspect of their development is to ensure that, besides being fast in the common case, these applications react fast to changes in their membership, e.g., due to reconfiguration and failures. This is especially important as they form the backbone of numerous cloud-powered services, such as analytics and trading systems, trying to meet ever-stringent tail latency requirements. As the microservices-oriented architecture is the de facto standard for building cloud services, a single user request translates to a wide fan-out of microservices interactions sitting on the critical path. The outcome is implacable: the traditionally uncommon events of reconfiguration and failures are exacerbated by the fan-out of communication, making user requests commonly experience such events and quickly impacting the tail latency of the service.

We present uKharon, a microsecond-scale membership service that detects changes in the membership of applications and lets them failover in as little as 50 μ s. uKharon consists of (1) a multi-level failure detector, (2) a consensus engine that relies on one-sided RDMA CAS, and (3) minimal-overhead membership leases, all exploiting RDMA to operate at the microsecond scale. We showcase the power of uKharon by building uKharon-KV, a replicated Key-Value cache based on HERD [24]. uKharon-KV processes PUT requests as fast as the state-of-the-art and improves upon it by (1) removing the need for replicating GET requests and (2) bringing the end-to-end failover down to 53 μ s, a 10 \times improvement.

1 Introduction

State-of-the-art data centers form the backbone of today's online services, including social networks, search engines, video streaming, e-commerce and banking platforms. The ever-increasing popularity of online services and their pervasive role manifest in both huge-scale requirements as well as stringent tail latency to guarantee smooth user interaction.

The tail of a cloud service refers to the latency of the slowest requests, and thus provides a limit to the maximum latency experienced by the end user. Despite substantial efforts in both hardware (e.g., InfiniBand/RDMA [40], RoCE [4], FPGA [6], Gen-Z [28], CXL [50]) and hardware-accelerated software [15, 21–23, 38, 52, 53, 55], keeping the tail short at large scale is one of the most important challenges in the cloud computing industry.

Dean *et al.* [9] shed light on the challenge of building tail-tolerant software at data center scale. This challenge mainly stems from the architecture of modern online services, which are composed of a plethora of layers that communicate frequently. Despite the scalability and cost benefits of such architectures, each end-user request results in a wide fan-out of interaction across tiers, each of which lies in the critical path between the service and its reply to the user. The probability of the traditionally rare reconfiguration and failure events is thus multiplied by the fan-out of the communication. As a result, user requests encounter such events more frequently, which quickly impacts the tail latency of the services.

Existing systems are not capable of handling failures within microseconds. Key-Value stores like Hermes [26], state machine replication [44] systems like Mu [2] and Hovercraft [29], and transactional systems like FaRM [12], process requests in a few microseconds in failure-free scenarios, but miss the microsecond envelope when handling failures. Mu and Hovercraft take 0.5ms and 10ms respectively to failover. Aguilera *et al.* [2] reported that Hermes has a failover of 150ms, while FaRM mentioned ZooKeeper [20], a widely used distributed coordination service that offers at-best millisecond failover, for its membership management.

This paper builds on the observation that a crucial step in making tail-tolerant microsecond applications is reacting fast to failures. We thus propose uKharon¹, a membership service tailored to the microsecond scale. Apart from acting as a distributed membership storage for (distributed) applications, uKharon monitors their nodes, detects their failures and

¹“u” stands for microsecond, and Kharon is the carrier of the souls of the dead in Greek mythology. It is pronounced ma · ka · ron.

changes their membership within 50 μ s. When uKharon itself experiences a failure, it recovers within 64 μ s. uKharon particularly benefits applications with efficient state transfer which can swap a faulty replica with a hot one in microseconds, for example via shadow replication. It targets cloud services that require seamless reconfiguration for fault tolerance and scalability, such as indexes, datastores and transactional systems.

The key to the performance of uKharon is the careful design of three fundamental components, all of which leverage RDMA to operate at the microsecond scale. *First*, uKharon achieves microsecond failure detection by employing a multi-level failure detector. It distinguishes the failures related to the application (e.g., segmentation faults), from those related to the kernel (e.g., driver faults), and failures related to the hardware (e.g., RDMA NIC faults), employing for each a different failure detector. *Second*, uKharon decides on memberships using a consensus engine which solely relies on one-sided RDMA verbs. This engine takes advantage of RDMA Compare-and-Swap (CAS) to handle leader changes within 10 μ s. *Third*, uKharon provides membership leases that add minimal overhead to the end application and last \sim 20 μ s. As a result, our membership service combines typically opposing forces: having applications with low-overhead dynamicity in failure-free scenarios and very fast failover upon failures.

We showcase the benefits of our membership service by building uKharon-KV, a replicated in-memory KV-cache based on HERD [24]. It uses uKharon to track the set of nodes and react to node failures. We compare uKharon-KV against HERD+Mu [2] (i.e., HERD replicated by Mu), a system which—to the best of our knowledge—achieved the lowest replication latency to date. Our evaluation shows that uKharon-KV processes PUT requests as fast as HERD+Mu in failure-free periods. Moreover, thanks to its leasing mechanism, uKharon-KV manages to spare the replication of GET requests, an optimization that is algorithmically impossible in HERD+Mu. As a result, uKharon-KV GETs are 31.8% faster than HERD+Mu’s. uKharon-KV, though, shines in the event of failures, achieving an end-to-end failover of 53 μ s, improving on HERD+Mu’s failover of 531 μ s by up to a factor of 10.

In a nutshell, we present uKharon, the first ever membership service suitable for the needs of tail-tolerant microsecond applications. We make the following contributions:

- A multi-level failure detector for the microsecond scale.
- A consensus engine that relies on one-sided RDMA CAS to change leader within microseconds.
- Microsecond leases that have minimal impact on the performance of the end application.
- uKharon-KV, a replicated KV-cache which outperforms the previous state of the art.
- The source code of uKharon is available at <https://github.com/LPD-EPFL/ukharon>.

The rest of this paper is organized as follows: Section 2 introduces background concepts. Section 3 gives an overview of uKharon’s design. Sections 4, 5 and 6 discuss the failure detection, consensus and leasing components, respectively. Section 7 reports on the performance of uKharon. Finally, Section 8 discusses related work and Section 9 concludes.

2 Background

2.1 Membership Service

To achieve resilience, long-lived distributed systems must be dynamic. Many systems [30, 31, 39, 45, 47] achieve dynamicity by relying on a coordination substrate, such as ZooKeeper [20] or etcd [14]. Among the various services (e.g., atomic locks, registers) these substrates offer, dynamicity is fundamentally addressed via their *membership service*.

A membership services offers dynamicity both in graceful executions and upon failures. In the former case, it serves join and leave requests issued by processes that want to become part of a distributed application or exit it. In the latter, it detects process failures and reacts to them. All these events are reflected through new configurations (called views or simply memberships). Essentially, a membership service acts as a storage of configuration information, keeping track of how the set of processes evolves, and exposes this information.

Typically, membership services rely on consensus [16] to establish a totally ordered sequence of views. Such services, including Zookeeper and etcd, offer strong semantics as all processes using the membership service transition through the same sequence of views.

Consensus-based membership services also offer real-time semantics. Apart from knowing the sequence of memberships, it is also important to know which is the (single) *active* membership. To understand why this real-time property is useful, consider the following example that incorrectly builds a cache storage solely relying on the sequence of memberships: The cache serves READ and WRITE requests. Initially, membership $M_1 = \{S_1\}$ designates server S_1 as responsible for the cache (i.e., S_1 stores it and serves requests). Eventually, a second membership $M_2 = \{S_2\}$ replaces S_1 with S_2 . S_2 , being part of M_2 , proceeds with serving clients’ requests and updates the content of the cache. At the same time, S_1 is unaware of M_2 and continues serving clients’ requests as well. As a result, a client that is also unaware of M_2 and reads from S_1 will get stale data. This example demonstrates a violation of consistency. It shows that total order of memberships does not provide any real-time guarantees by itself.

Membership services provide real-timeness by making outdated memberships nonoperational. A commonly used mechanism to achieve this property is the use of a distributed invalidation protocol. Another solution is to rely on leases. With leases, processes are forced to periodically check the active membership, execute operations in this membership,

and abort operations that span over multiple memberships. uKharon provides real-timeness via leases.

2.2 RDMA

Remote Direct Memory Access (RDMA) [49] is a networking technology that allows processes to access the memory of a remote machine without involving the CPU of the latter. By implementing several layers of the networking stack in hardware and relying on kernel bypass, RDMA achieves microsecond inter-machine communication. It allows applications within the data center to communicate in as little as $0.9\mu\text{s}$ [25]. This technology is supported by different fabrics such as InfiniBand [49] and commodity Ethernet via RoCE [4].

Applications communicate over RDMA by relying on primitives called *verbs*. There exist *one-sided verbs* that include READ, WRITE and Compare and Swap (CAS) verbs and *two-sided verbs*, such as SEND and RECV verbs. One-sided verbs let a process read, write and apply atomic transformations to a remote machine’s memory without involving its CPU. Two-sided verbs are similar to message passing and involve both communicating sides. They let processes send and receive memory buffers. Communication in RDMA can notably occur over established *Reliable Connections (RCs)* or over *Unreliable Datagrams (UDs)*. While the former provide FIFO semantics, the latter trade reliability for better performance and support for message multicast [49].

2.3 Communication Model

uKharon is designed for data centers. It is safe under asynchrony and live under partial synchrony [13]. That is, to make progress, uKharon assumes a Global Stabilization Time (GST), unknown to the processes, such that from GST onwards there is a bound Δ on communication and processing delays. This is a realistic assumption, as data center fabrics are not asynchronous in practice [3, 35, 54]. Additionally, our system relies on bounded clock drift for safety, i.e., durations are approximately the same across all processes. uKharon also assumes *crash-stop* failures: processes may fail by crashing, after which they stop executing. Finally, we assume that network partitions, which affect uKharon’s liveness, are eventually resolved by the data center administrators.

3 Design Overview

3.1 Architecture

Figure 1 gives an overview of uKharon. Our system, as a membership service, runs on application nodes as well as a set of dedicated nodes called *coordinators*.

Central to uKharon is uKharon Core, a single-threaded library that hosts monitoring functionalities of the membership service. This includes detecting failures of member nodes

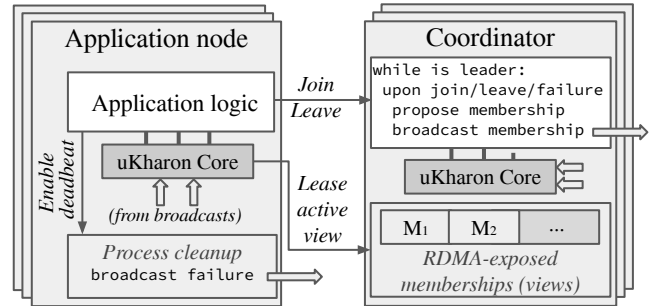


Figure 1: Overview of uKharon

(including coordinators), listening for failures and new memberships, as well as renewing leases. The application receives these events via thread-safe accessors: a stream of failures, a stream of memberships and a method `Active(M) → bool` which checks whether a given membership M is active.

The generation and storage of memberships is delegated to coordinators. Coordinators achieve fault tolerance through consensus. One of them is the leader, which processes join/leave requests from both application nodes and coordinators, proposes new memberships and broadcasts decided memberships which are picked up by the uKharon Core instance running on every node. The rest of coordinators help the leader decide and replicate the sequence of memberships. Finally, coordinators assign each member a unique identifier.

Running uKharon Core on both application nodes and coordinators helps bootstrap the membership service. uKharon Core learns about the new memberships from coordinators, but coordinators require the membership service to learn about each other. Similarly, coordinators rely on uKharon Core to detect failures of application nodes or themselves.

Part of uKharon’s failure detection logic resides in the kernel, outside of uKharon Core. It consists of a kernel module hooked to Linux’s process cleanup routine. This module can be enabled by the application logic and broadcasts a failure notification (called *heartbeat*) when the application crashes.

New memberships are merely broadcast by coordinators, putting the burden of detecting the active membership to the application nodes. uKharon Core is responsible for bringing real-timeness to applications. It reads the RDMA-exposed memberships at a majority of coordinators to determine whether a membership has been superseded by a new one or whether it is still active. The active membership is leased for a limited amount of time, in our case $\sim 20\mu\text{s}$.

3.2 Communication

uKharon relies extensively on the performance of today’s RDMA-enabled fabrics to achieve its microsecond latency target. It leverages one-sided RDMA verbs, two-sided ones (i.e., HERD-style RPC [24]), as well as RDMA Multicast. Coordinators run consensus using RDMA Reliable Connections

(RCs). In particular, coordinators establish all-to-all connections among themselves and communicate using RDMA READ, WRITE and CAS. Additionally, coordinators use RDMA Multicast, which is backed by RDMA Unreliable Datagrams (UDs), to notify all nodes about new memberships. uKharon also uses RDMA Multicast to emit failure notifications. uKharon Core relies on RDMA READs over RCs to retrieve the active membership from coordinators and to detect the failure of remote nodes. Finally, processes send *join* and *leave* requests to the coordinator leader using RPC.

3.3 Challenges

Our system is designed for applications that operate and failover at the microsecond scale. To do so, uKharon meets two important design goals. First, it itself operates at the microsecond scale, meaning that it is able of changing the active membership within as few as 50 μ s. Second, we ensure that uKharon Core has minimal performance overhead on the end application it is bundled with. To meet these goals, uKharon is structured around three major components:

Failure detection. Efficient failure detection is the first step towards fast failover. Conventional wisdom suggests that there is a trade-off between the speed and accuracy of a failure detector. We work around this limitation by building a hierarchy of RDMA-tailored failure detectors suited for the microsecond scale. Our hierarchy detects failures within a few tens of microseconds, as we explain in Section 4.

Consensus engine. The second step of failover is agreeing on the new membership. Existing leader-based consensus engines, although optimized for the microsecond scale, struggle to change their leader at this time scale. In Section 5, we explain how our microsecond consensus engine changes leader in microseconds. This gives our design the unique property that a coordinator failure—especially failure of the coordinator leader—has negligible effect on the failover time.

Leases. As far as the membership service is concerned, the last step towards failover is updating the active membership. However, the new membership cannot become active before leases on previous memberships have expired. Thus, the longer the leases, the higher the failover time. On the other hand, short leases can result in application overhead, as they have to be checked in the application’s critical path and renewed in time before expiring. In section 6, we explain how uKharon manages to have $\sim 20\mu$ s leases with virtually no cost for the end application and how leases can scale to hundreds of machines for an extra $\sim 20\mu$ s.

4 Microsecond Failure Detection

uKharon relies on microsecond failure detection to notify nodes about member failures and to trigger the generation of

new memberships. In this section, we describe uKharon’s failure detection scheme.

4.1 Multi-level Failure Detection

A practical failure detector aims at being as complete and as accurate as possible. A complete and accurate failure detector is able to detect all failures and not have false positives, respectively. Completeness without accuracy causes problems in practice, as false positives trigger new memberships which require distributed applications to take further action (e.g., rebalancing data among nodes).

Commonly, failure detectors rely on timeouts for their operation. However, timeouts are hard to set correctly: if they are too low, the failure detector may experience instability (e.g., oscillating behaviors). That explains why most systems set the timeouts to a safe high-enough value. In the microsecond scale this problem is magnified, as small execution delays (e.g., kernel jitter) can take several microseconds.

Our failure detector follows a pragmatic approach: it avoids timeouts when possible. To achieve this, we are inspired by Falcon [35], and identify four levels of failures: (1) *userspace failures* (e.g., segmentation faults, out of memory errors, uncaught exceptions) that cause the application to abort, (2) *kernel failures* (e.g., cores hanging in the kernel, kernel oops caused by driver crashes) that impede the application’s execution, (3) *catastrophic failures* (e.g., power failures, RDMA NIC failures) that prevent communication with the application’s host, and (4) *byzantine failures* (e.g., stack overflows, mercurial cores [19]) that affect the application state. Each of the first three levels is handled by uKharon via a specialized failure detector. We do not address Byzantine failures.

4.2 uKharon’s Failure Detectors

We now explain how uKharon’s specialized failure detectors work, depending on the type of failure.

Userspace failures. They are handled by the Linux kernel. The application registers to the kernel to enable a *deadbeat*, which is a failure notification broadcast by the kernel upon the death of the process. This registration happens by means of the `prctl` system call that the application calls early in its execution. The system call includes the node’s identifier and modifies the process descriptor (Linux’s `task_struct`) with a flag that the kernel checks during the *cleaning routine* of the process. In Linux, when a process crashes, control is transferred to the kernel which starts executing the process cleaning routine. If the flag is set, the kernel broadcasts a failure notification that includes the specified identifier. To achieve this functionality, we extend the `prctl` system call and modify the process cleaning routine that is part of the kernel’s `exit` system call. The task of broadcasting the crash notification is delegated to a kernel module. This module uses the kernelspace RDMA driver to broadcast crash notifications

which are polled by all instances of uKharon Core. As this failure detector does not use timeouts, it has no false positives.

Kernel failures. To detect application failures caused by the kernel, we rely on the way RDMA is handled in userspace. An application registers memory to an RDMA device by issuing `ioctl` system calls on a file descriptor. By design, the Linux kernel destroys that file descriptor and thus disables remote access to this memory at the end of the process' cleaning routine. If this cleaning routine runs, the failure is caught by the previous failure detector. Otherwise, the memory will remain remotely accessible while the execution of the application is suspended (and the kernel is dying).

For the operation of this failure detector, processes are arranged in a logical ring where every process monitors its successor. Our system uses a local heartbeat counter in a similar fashion to Mu's detector [2]. uKharon Core increments this counter to indicate that the process is alive. This counter is read by the predecessor process. If a process RDMA-reads the same value twice, it reports its successor as having failed.

A process would be wrongly detected if it were unable to increment its counter between two consecutive reads. Thus, we take special care to ensure that processes always increment their counters faster than the time delay between two consecutive reads. Importantly, we deploy (the single-threaded) uKharon Core in its own dedicated physical core. We resort to a custom kernel compiled with the `NO_HZ_FULL` option, which disables regular timer interrupts [37] on the dedicated core and thus reduces the kernel jitter towards uKharon Core. Additionally, we boot this kernel with the `isolcpus` parameter, which prevents other userspace processes from sharing the dedicated core with uKharon Core. In experiments, the interval we observed between two counter increments under heavy load was $5\mu\text{s}$ most of the time and never more than $15\mu\text{s}$. To account for unexpected jitter (e.g., thermal throttling), we make processes wait $30\mu\text{s}$ after the completion of an RDMA READ before issuing the next one. As RDMA READs are issued sequentially, network delays do not negatively impact the accuracy of this failure detector.

Catastrophic failures. uKharon relies on a timeout-based scheme to detect failures that prevent machines from communicating. We set the timeout to 1ms, which is 2 – 3 orders of magnitude higher than the common case latency of modern data center fabrics. As reported by Li *et al.* [36], 1ms is safe even in case of network congestion.

The detector works by having processes periodically broadcast a heartbeat and poll for heartbeats from others. Processes keep track of the set of processes they recently received a heartbeat from. They compare this set with the current membership and report which processes they consider failed to the coordinator leader. Then, the leader constructs a connectivity graph based on the reported link states and changes the membership to approximately match the maximum clique in which it is included. Thus, our membership service en-

forces all-to-all connectivity among the members and does not expose any information regarding network partitions. A systematic treatment of network partitions is out of our scope.

The first two detectors broadcast failure notifications over RDMA-multicast, which offers better scalability than broadcasting using Reliable Connections. Nevertheless, RDMA-multicast is backed by Unreliable Datagrams, thus failure notifications can be lost under high network load. Dropping these notifications is safe, as uKharon-Core rebroadcasts a failure notification until a new membership excludes the failed node.

5 Microsecond Consensus

In this section, we present a state-of-the-art consensus engine that is tailored for the needs of uKharon and powers its coordinators. Our engine is efficient regardless of failures: in the absence of failures, it decides in one RDMA delay (by issuing an operation to a majority of processes in parallel), while it decides in one additional RDMA delay in the event of a failure. It uses a slightly modified version of Paxos based on the observation that the original algorithm contains RPCs that can be emulated with RDMA CAS operations. In the rest of the section, we intuitively describe our consensus algorithm and discuss implementation details. Appendix A provides its pseudocode and a proof of its correctness.

5.1 Consensus and Paxos

Consensus is a fundamental problem in distributed computing. Informally, each process proposes a value and eventually all processes irrevocably agree on one of the proposed values. Processes agree on a sequence of values and totally order them by running multiple instances of consensus.

Several algorithms solve consensus in the partially synchronous model. Many are variants of Paxos [32]. In Paxos, processes are divided in two groups: *proposers* and *acceptors*. Proposers *propose* a value for decision and acceptors *accept* some proposed values. Once a value has been accepted by a majority of acceptors, it is decided by its proposer.

Intuitively, Paxos is split in two phases: the *Prepare* phase and the *Accept* phase. During these phases, messages from the proposer are identified by a unique *proposal number*. The Prepare phase serves two purposes. First, the proposer gets a promise from a majority of acceptors that another proposer with a lower proposal number will fail to decide. Second, the proposer updates its proposed value using the accepted values stored in the acceptors. This way, if a value has been decided, the proposer will adopt it. The prepare phase can also *abort* if any acceptor in the majority previously made a promise to a higher proposal number. If the proposer manages to complete the Prepare phase without aborting, it proceeds to the Accept phase. In this phase, the proposer tries to store its value in a

```

1 # Paxos's RPCs pattern
2 def rpc(x):
3     if compare(x, state):
4         state = f(state, x)
5         return proj(state)
6
7 def cas_rpc(x):
8     expected = fetch_state()
9     if not compare(x, expected):
10        return proj(expected)
11    move_to = f(expected, x)
12    old = state.cas(expected,
13                  ↪ move_to)
14    if old == expected:
15        return proj(move_to)
16    abort

```

Algorithm 1: Paxos's RPCs turned into CAS-based RPCs.

majority of acceptors. If it succeeds (i.e., a majority accepted the value), it decides on that value.

5.2 One-sided Paxos

Paxos uses RPC in a very specific form. The acceptors' state consists of only three variables: `min_proposal`, `accepted_proposal` and `accepted_value`. In both phases, acceptors atomically update these values based on the proposer's input and return some of them.

Algorithm 1 proposes an obstruction-free transformation to turn Paxos's RPCs into purely one-sided conditional writes using RDMA CAS. Paxos's RPCs follow the pattern seen in `rpc`. The acceptor executing the RPC compares the received value `x` to its state (stored in `state`). If the comparison is successful, the acceptor updates its state (shown with function `f`) using the provided value `x`. Finally, the acceptor unconditionally returns part of its state (shown with function `proj`).

The pattern presented in `cas_rpc` allows RDMA to emulate `rpc` while solely relying on one-sided verbs. Opposite to `rpc`, which is executed on the acceptor's side, `cas_rpc` is executed on the proposer's side. To execute the one-sided RPC, the proposer first needs to know the `state` that is stored in the memory of the acceptor. This value can either be guessed (e.g., using a previous value of `state`) or fetched (e.g., using RDMA READ, as shown in line 2). Then, the proposer executes the comparison locally (line 3) and decides whether to continue or terminate. If the comparison succeeds, the proposer proceeds with updating the state of the acceptor. It is this update that utilizes CAS². In line 7, if the CAS succeeds, the acceptor's `state` has been updated successfully with the value of `move_to`. Otherwise, `state` remains unchanged.

When the RDMA CAS succeeds, i.e., in the absence of contention, both `rpc` and `cas_rpc` are equivalent (see Appendix A.2). However, if the RDMA CAS fails, `cas_rpc` will abort while `rpc` would not. In this case, `rpc` and `cas_rpc` are not equivalent, but this does not violate the correctness of Paxos. The reason is that Paxos tolerates an arbitrary number of proposer failures and that aborting the RPC and starting over is indistinguishable from such a failure.

²As a reminder, `variable.cas(expected, new)` atomically checks if `variable` equals `expected` and sets `variable` to `new` if this is the case. The operation always returns the initial value of `variable`.

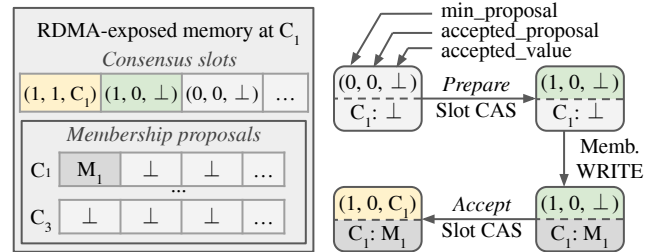


Figure 2: uKharon's Consensus Engine with its RDMA-exposed memory for multiple instances of consensus (left) and a state machine for a single instance of consensus (right).

5.3 uKharon's Consensus Engine

We now explain how to make the variant of Paxos described in Section 5.2 practical and compare it with Mu [2], a state-of-the-art consensus engine.

5.3.1 Practical Considerations

Leader election. To avoid the contention rising from multiple concurrent proposers, our consensus engine adopts the same leader election scheme as Mu. The process with the lowest identifier among the coordinators considered alive is elected as the leader. In the event of a partial network partition, this scheme can elect multiple leaders. For example, if coordinator C₂ is the only one unable to reach C₁, it will think of itself as the leader, while other coordinators will consider C₁ as their leader. Having multiple leaders cannot lead to multiple values being decided, i.e. safety is always preserved. Leader contention can, however, prevent the engine from being live. Thus, a leaders that fails to decide uses a randomized backoff before proposing until the partition is resolved.

Pre-preparation. Coordinators decide on a sequence of values by running consensus on a sequence of *slots*, as shown in Figure 2. It requires two RDMA delays for each slot: one for the Prepare and another for the Accept phase (shown with horizontal arrows in the figure). A stable leader can prepare slots in advance and only run the Accept phase to decide. In this case, the leader decides in a single RDMA delay. The leader uses the time spent waiting for the Accept phase to complete on a slot to run the Prepare phase for the next one. Thus, it always maintains one pre-prepared slot (depicted in the second consensus slot of Figure 2), with no latency overhead. Switching to the new leader requires re-preparing the next slot. As an optimization, the new leader predicts that the last slot had been prepared by the previous leader and uses this prediction as the expected value of the RDMA CAS. With this approach, the new leader manages to re-prepare the next slot in a single RDMA delay instead of two.

CAS size limitation. Algorithm 1 assumes that the consensus state fits within a single CAS. Current RDMA NICs only

support CAS up to 8 bytes. We set both `min_proposal` and `accepted_proposal` to be 2 bytes each³. The remaining 6 bytes are dedicated to the `accepted_value`.

Our consensus engine uses indirection to overcome the limited size of the `accepted_value` and store uKharon's memberships. Instead of deciding on the membership itself, coordinators decide on its location in memory. First, the proposer RDMA-writes the membership to a part of acceptors' memory dedicated to membership proposals (see Figure 2) to which it has exclusive write access. Then, the proposer runs the Accept phase where it proposes its own identifier (C_1 in the figure). If the Accept phase succeeds at a majority of acceptors, then the proposer decides. Thanks to the FIFO semantics of RDMA RCs, if the last RDMA operation (i.e., the Accept phase CAS) succeeds, the previous RDMA operation (i.e., storing the membership with an RDMA WRITE) also succeeded. The two RDMA operations combined do not execute atomically, yet a coordinator cannot have accepted an identifier without knowing its associated membership.

5.3.2 Comparison with the State-of-the-art

Many systems, such as Mu [2], DARE [41] and APUS [51] study consensus over RDMA. They primarily focus on improving the throughput and latency of common case executions, thus achieving consensus in a few microseconds. However, these systems have failovers ranging from 0.5ms (in Mu) to 10s or 100s of ms (in DARE and APUS, respectively).

Mu has the best performance in failure-free executions among competition as it solves consensus in $\sim 1.4\mu\text{s}$. It relies extensively on RDMA permissions. During its Prepare phase, a proposer asks acceptors for the exclusive write permission to their memory and waits for a majority of replies. This step guarantees that only one proposer can write to an acceptor at a time. In the Accept phase, the proposer decides by merely writing to a majority of acceptors. As acceptors give write permissions to a single proposer at a time, no two concurrent proposers can successfully write to a majority of acceptors and decide on different values. Since WRITE is the most efficient RDMA verb and the Prepare phase runs only once per leader change, Mu is optimal in failure-free executions.

The Accept phase of our algorithm relies on a WRITE followed by a CAS. Importantly, these one-sided operations have lower tail latency compared with the two-sided verbs present in DARE and APUS. The CAS increases the decision time from $1.4\mu\text{s}$ to $2.9\mu\text{s}$ compared with Mu. When it comes to a leader change, Mu's permission change mechanism requires approximately $250\mu\text{s}$, since it constitutes a control path operation that involves a system call and a reconfiguration of the NIC. In our consensus engine, the additional CAS lets coordinators change leader in under $10\mu\text{s}$. Thus, our algorithm is designed for short tail latency and makes the failure of the

³Appendix A.6 discusses how to prevent overflows after 2^{16} failed attempts to decide on a slot by switching from CAS-based to two-sided RPCs.

coordinators' leader no more important (latency-wise) than the failure of any other node.

6 Microsecond Real-timeness

In addition to reacting to failures and deciding on views, uKharon lets applications track the active membership via the `Active` method. While this information is essential for consistency, it must not burden the end application. In this section, we describe the challenge of making `Active`'s overhead negligible while preserving microsecond view changes.

6.1 The Active Method

uKharon exposes real-timeness to end applications via the `Active(Membership) → bool` method. If `Active(M)` returns `true`, we say that M is *active* at some point between the call and return of the method. `Active` satisfies three important properties. First, there are no two overlapping active memberships. Second, after a membership M is active, no memberships older than M become active. Third, the active membership converges to the latest decided membership.

Intuitively, processes use the `Active` method to determine the membership they should be executing operations in. When coordinators decide on a new membership M' , a process p may stay in an older membership M due to a delay in receiving M' . Calling `Active(M)` will eventually return `false` at p , thus letting it realize that it misses the latest membership M' . To ensure consistency, an application typically calls `Active` once before starting an operation and a second time before committing it, only committing if both calls return `true`.

6.2 Leases

uKharon uses leases for efficiency. We proceed incrementally, first describing an implementation of `Active` without leases, before moving to a more efficient lease-powered scheme.

The basic implementation of `Active` requires communication in every invocation. Let M be the k -th membership decided by the coordinators and assume a process p invokes `Active(M)`. In essence, `Active` declares that M is active if it can conclude that no newer membership M' has been decided. To this end, the process RDMA-reads the $k + 1$ -th consensus slots at coordinators and waits for a majority of replies. If all replies are empty, then the $k + 1$ -th membership has not been decided, meaning that M is (still) active at some point between the invocation and return of the method. If, on the other hand, at least one of the replies is non-empty it is inconclusive whether M has been superseded by M' . In case M' has been decided before p issues the READs, then at least one of the replies must be non-empty, but the opposite is not always true. For safety, `Active` returns `false` if at least one of the READs on the next consensus slot is non-empty.

```

1 leased_membership = ⊥; t_start = 0; t_end = 0
3 def Active(M) → bool: # M is always a decided membership
4   t = hw_timestamp()
5   if leased_membership != M: # First-time lease on M
6     if majority_active(M):
7       leased_membership = M; t_start = t + δ; t_end = t_start
8   else: # Check/extend lease on M
9     if t in [t_start, t_end): return True
10    if majority_active(M):
11      t_end = t + δ
12    return t > t_start
13 return False

```

Algorithm 2: Leased active membership.

A lease refers to a membership and has a start and an expiration date. A lease guarantees its holder that its associated membership will remain active until it expires. In our system, leases are created by uKharon Core and last $\delta \approx 20\mu\text{s}$.

Algorithm 2 provides an efficient alternative implementation of `Active` that relies on leases to reduce communication. It starts by taking a hardware timestamp t (line 4) and then checks if a lease on M already exists (line 5). If no lease exists (lines 6-7), the method checks for a newly decided membership by contacting a majority of coordinators. If no membership newer than M could have been decided (i.e., all replies are empty), it creates a lease on M (line 7) that starts at $t + \delta$ and has no duration. This prevents overlapping active memberships since any lease that processes could hold on a previous membership $M' < M$ will have expired before M becomes active. In case a lease on M already exists, the method tries to use it in order to avoid reaching the coordinators (line 9). If it cannot use it, it tries to extend the lease (line 11) by checking the coordinators. It returns `True` only if leases on previous memberships have expired (line 12), which takes—in the worst case— δ to happen. As a result, leases affect the speed at which memberships can change, justifying the desire for a small lease duration. Section 7 demonstrates that leases of $\delta \approx 20\mu\text{s}$ are feasible in practice.

This efficient implementation of `Active` renews its lease on demand. As long as its lease is valid, the method merely takes a hardware timestamp—which takes a few tens of nanoseconds—and returns immediately without reaching the coordinators. The latency overhead of `Active` to the application that invokes it is thus very low. Communication with the coordinators is only necessary when leases expire and have to be renewed, which results in a spike in `Active`'s latency. In practice, uKharon Core renews leases in the background to ensure that—when the membership remains unchanged—`Active` is not delayed by the calls to `majority_active`.

uKharon does not rely on operational leases for either liveness or safety. Timely renewal of leases is only a way to reduce the latency of `Active` as Algorithm 2 would work even with zero-duration leases. uKharon relies on bounded clock drifts for safety, as opposed to clock synchronization. This ensures that durations are approximately the same across

all processes, thus preventing overlapping memberships. Appendix C includes a microbenchmark evaluating the clock drift of actual hardware and gives an overestimated drift that is no more than 0.001% of the lease duration. Thus, clock drift is accounted for by making leases last a few nanoseconds less than their nominal value. As drift is reset on each lease renewal, it does not accumulate over time. Therefore, no matter how long a system is up for, its operation remains unaffected by the clock drift. A proof of correctness of uKharon's leases is given in Appendix B.

6.3 Extensions

Adaptive leases. So far, we have assumed a fixed lease duration δ . Network delays greater than δ render leases useless as, every time the lease is extended (line 11), t_{end} is always in the past. In this case, `Active` always contacts the coordinators. In order to work under partial synchrony and avoid this scenario, we extend the leasing mechanism as follows: Coordinators store the lease duration for a given membership along with the membership itself. An application node that wants to increase the lease duration contacts the coordinator leader. This results in a new *compatible* membership that is identical to the previous one apart from the lease duration. Compatible memberships receive special handling by uKharon Core in order to ensure that—when going from one compatible membership to another—`Active` does not wait for leases on the previous membership to expire. Also, if the latest membership M is not compatible with the previous one, invocations to `Active(M)` return false until all possibly ongoing leases on previous memberships have expired.

Lease caches. `Active` reaches a majority of coordinators to renew its lease, which scales badly as the number of application nodes increases. uKharon solves this issue with an intermediate lease renewal layer, the *lease caches*. These caches use the `Active` method to lease memberships for Δ (by reading from a majority of coordinators). In turn, application nodes use leases that last for δ and a modified version of `Active`. This version differs from the one presented in Algorithm 2 in the `majority_check` calls, which are replaced with RPCs to a *single* lease cache. As a result, application nodes reduce the communication cost required to renew their lease by a factor of—at least—3 (the typical number of coordinators). However, lease caches increase the failover time of applications by at least Δ . The reason is that when the coordinators change the membership, the `Active` method of caches waits Δ before making the new membership active. At the same time, the `Active` method of application nodes that is directed to some lease cache, waits δ before making the new membership active. Thus, the overall time from the moment a new membership is decided until application nodes start using it jumps from (at least) δ to (at least) $\Delta + \delta$.

7 Evaluation

We evaluate the various performance traits of uKharon and verify its suitability as a membership service for microsecond applications. We aim to answer the following:

- How much does uKharon increase the latency of end applications and what is its impact on their throughput?
- How fast does uKharon respond to failures?
- How can uKharon be leveraged to build replication protocols and what performance can they achieve?

CPU	2x Intel Xeon Gold 6244 CPU @ 3.60GHz (8 cores/16 threads per socket)
NIC	Mellanox ConnectX-6 MT28908
Switch	Mellanox MSB7700 EDR 100 Gbps
OS/Kernel	Ubuntu 20.04.2 / 5.4.0-74-custom
RDMA Driver	Mellanox OFED 5.3-1.0.0.1

Table 1: Hardware details of machines.

We evaluate uKharon in a 8-node cluster, the details of which are given in Table 1. The custom kernel sets the `NO_HZ_FULL` option and uses the `isocpus` boot parameter, as explained in Section 4.2. Our dual-socket machines are NUMA and their RDMA NIC lies on the first socket. For this reason, we ensure that all threads during our experiments execute on cores of the first socket. We also make all threads exclusively use the memory bank closest to this socket.

Our implementation measures time durations using the `clock_gettime` function with the `CLOCK_MONOTONIC` parameter. The function uses the TSC clocksource of the Linux kernel, which offers efficient and accurate timestamping [43]. Appendix C discusses details regarding the drift and synchrony of TSC in symmetric multiprocessing (SMP) systems.

Finally, in all experiments we deploy 3 coordinators.

Applications. We integrate uKharon with HERD [24]. HERD is a non-replicated microsecond-scale RDMA-based KV-cache. Clients send requests to a HERD server by RDMA-writing to a dedicated buffer that the server has allocated for them. Requests contain an 8-byte key and are either PUTs or GETs. PUTs additionally contain the value to be stored for the specified key. The server discovers new client requests by polling its local memory, executes the requests locally and then replies to the clients using RDMA UD. We also leverage uKharon to build uKharon-KV, an extended version of HERD which supports replication. We compare our solution with HERD replicated by Mu (HERD+Mu) [2] which—as far as we know—offers the lowest replication latency to date.

Implementation effort. We implemented uKharon on top of our own RDMA framework. uKharon Core and the consensus engine span 4448 and 1324 lines of C++, respectively. The

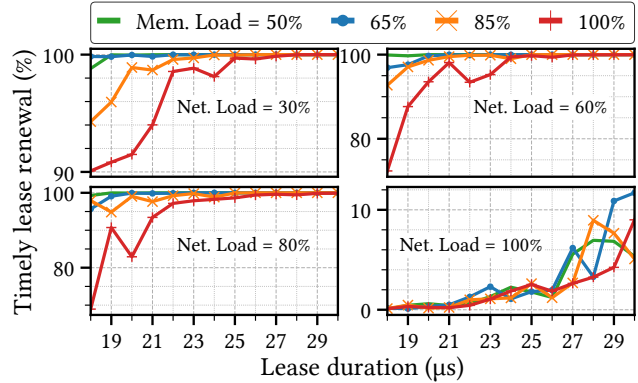


Figure 3: Percentage of timely lease renewal depending on the lease duration, network load and memory load.

kernel module of the deadbeat failure detector is 404 lines of C. uKharon-KV extends HERD by 1498 lines of C++. The only unimplemented features are clique-based memberships (Section 4.2) and adaptive leases (Section 6.3).

7.1 Overhead Induced by uKharon

Latency Overhead. Applications bundled with uKharon Core rely heavily on its `Active` method. As long as (the background running) uKharon Core renews the lease on the active membership in time, the `Active` method adds negligible latency overhead to the application. We experimentally determine that the 99th percentile latency for invoking `Active` is 38ns when the lease is renewed in time, which is the time it takes to fetch the hardware timestamp and compare it with the expiration date of the lease. Fluctuations in the network’s latency or execution delays when uKharon Core renews the lease (e.g., due to cache misses) induces additional latency to the application, as explained in Section 6.2.

Figure 3 shows how the duration of leases affects their timely renewal. We run 1-minute experiments under a steady membership with 32 lease renewers contacting coordinators directly and lease durations ranging from 18 to 30 μ s. Each machine has a maximum memory bandwidth of 480Gbps and a maximum network bandwidth of 100Gbps. We apply variable network and memory load by running `stress-ng` [27] and `perftest` [42] on the first socket of our machines.

When the network load is maximum (bottom right figure), less than 12% of the calls to `Active` return immediately, irrespective of the memory load. For network loads of 30–80% (other figures), the memory load progressively affects lease renewal. Maximum memory load causes expired leases when lease duration is shorter than 27 μ s. For most other configurations, a duration greater than 23 μ s suffices. For example, with 80% network and 50% memory load, lease renewal fails 0.0011% of the time, which corresponds to `Active` inducing latency every 300 out of 1.5 billion invocations. In other

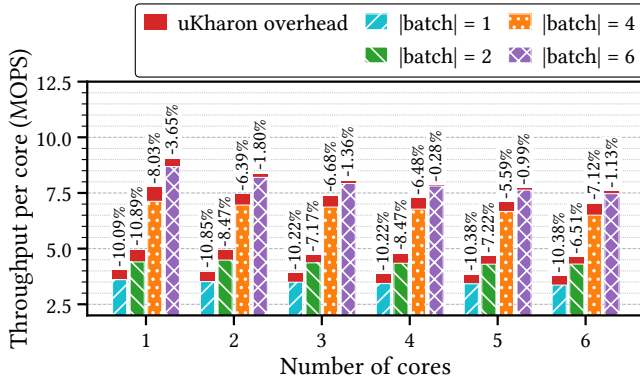


Figure 4: Impact of uKharon on HERD’s throughput for different batch sizes and numbers of cores. Full bar shows the throughput w/o uKharon; labels show uKharon’s overhead.

words, the 99.999th percentile of Active’s latency is 2 μ s.

We get similar (omitted) results when an application renews its leases through lease caches. In fact, RPC-based renewal requires at most 2 μ s longer leases (compared with reading from coordinators) to achieve the same percentages of timely lease renewal. We attribute this difference to RPC, which involves the CPU of both the application and the lease cache.

From this experiment we select the lease duration that we use for the rest of our evaluation. We pick the lease duration when renewing from coordinators (δ) to be 23 μ s, and the lease duration when renewing from lease caches (Δ) to be 25 μ s.

Throughput Reduction. We use uKharon to make HERD dynamic. The original HERD assumes a static set of servers, each of which serves a shard of the key space. Clients are aware of this sharding and use the key of a request to determine the appropriate server. The lack of dynamicity affects HERD’s flexibility in two ways. First, if a server fails, its shard becomes unavailable forever. Second, the system is unable to re-balance the load among the servers. Importantly, the use of a static set of servers ensures consistency of clients’ requests: GETs return the value of the most recent PUT.

In our implementation, each server dedicates up to 6 cores to the KV-cache and each core is responsible for a part of the key space. Every core processes clients’ requests and invokes the Active method before replying to avoid inconsistencies. If Active returns true, the core executes the request (if the key belongs to its shard) and replies to the client. Otherwise, the core rejects the request. Given that every core invokes Active in the critical path of serving requests, the latency of requests increases (by \sim 38ns) and the throughput decreases.

Figure 4 shows the per-core throughput of a static deployment of HERD, along with the drop in performance caused by the integration of the Active method. The workload is 80% GETs and 20% PUTs with 32 byte-long values. We vary the number of cores from 1 up to 6 as well as the batch size (i.e., the number of clients’ requests processed at once). Typically,

static HERD issues a reply every 350ns. Without batching, having Active in the critical path raises the reply time to 388ns, an increase of 11%. Batching has a positive impact on Active’s overhead as a single call to the method is used to serve all the requests in a batch. Thus, for batches of 6 replies, Active effectively takes 38/6 = 6.3ns per reply, an increase of just 1.8%. Finally, the overhead of Active does not increase with the number of cores, even though they invoke the method concurrently. This indicates good multicore scalability, which implies that a single uKharon Core instance per server is sufficient to serve all applications running on it.

Bandwidth overhead. uKharon Core reduces the bandwidth available to applications. Lease renewal requires 240 bytes when contacting 3 coordinators and 132 bytes when contacting a lease cache, which translates to (assuming renewal every 10 μ s) 192Mbps and 105Mbps, respectively. This bandwidth requirement accounts for 0.1 – 0.2% of a 100Gbps link, thus the bandwidth of application nodes is marginally impacted. Failure detection has similar bandwidth requirement.

7.2 Failover Time

We study uKharon’s failover time considering userspace and kernel failures. We do not further evaluate catastrophic failures, as 95% of the failover is for their 1ms-long detection, making microsecond-scale agreement and leases insignificant.

Table 2 summarizes the median failover (over 100 measurements) for various failure scenarios. We consider the failure of a single application node optionally combined with the failure of the coordinator leader or/and a lease cache. We emulate simultaneous failures by relying on RDMA Multicast. An auxiliary program executes alongside the program which we emulate the failure of. When the auxiliary program receives the multicast message, it uses SIGKILL to kill the targeted program. We assume the worst scenario, i.e., the failure of the application node results in global unavailability that is resolved only by a new (active) membership that excludes it.

In every entry of Table 2, we present the failover time when detecting the failure using the deadbeat mechanism (left) and the RDMA-based heartbeat mechanism (right). We now discuss the failover time when using the deadbeat, first considering the case when the lease caches are absent. For a single application failure, uKharon is able to failover in 50 μ s using the deadbeat. If the coordinator leader crashes at the same time as the application, the failover time increases by around 15 μ s. We attribute this increase to (1) the leader switch mechanism of the consensus engine (\sim 10 μ s) and (2) the imperfect synchronization of SIGKILL among the failed nodes (\sim 5 μ s). When lease caches are part of uKharon, the failover times for the same failure scenarios increase (as expected) by 20 – 25 μ s, which is about the lease duration of the cache. Failure of a cache has no impact on the failover time (bottom entries of the first and third columns). This is because (1) the application node receives the broadcast failure

L exists?	A	A + C	A + L	A + L + C
No	50\96	64\114	-	-
Yes	74\108	96\138	75\113	101\139

Table 2: Failover time (in μs) for failures in App, Coordinator leader and Lease caches; using the `deadbeat\heartbeat`.

notification and switches lease cache before the membership changes and (2) the new membership is compatible with the previous one. The simultaneous failure of all three types of nodes has a downtime of $101\mu\text{s}$, instead of $96\mu\text{s}$. Again, the failure of the cache does not affect the failover time, but with three nodes the imperfect synchronization of failures adds up. Finally, the same failures when using the RDMA-based heartbeat mechanism range from 96 to $139\mu\text{s}$. This mechanism adds $\sim 45\mu\text{s}$ of failover compared to the deadbeat. The reason is that reading the same value twice upon failure takes 1.5 delays on expectation and READs are issued every $30\mu\text{s}$.

7.3 uKharon-KV

Both uKharon-KV and HERD+Mu follow a primary-backup replication scheme. All requests are served by the primary, which replicates them to backups. Backups are only used for fault tolerance. All replicas (primary and backups) execute requests in the same order, but only the primary replies to clients. In the event of a failure of the primary, one of the backups becomes the new primary and continues serving clients' requests. All replicas execute all requests in the same total order, thus replicas are an exact copy of the failed primary. This means that when a replica becomes the new primary, it can respond to clients without breaking consistency.

One problem these systems have to deal with is multiple nodes trying to replicate clients' requests simultaneously. This happens when the primary fails and multiple nodes, believing they are the new primary, try to handle clients' requests. Mu avoids this problem by relying on RDMA permissions (see §5.3.2). On the other hand, uKharon-KV relies exclusively on the membership service to address it. Each membership determines a single primary. When the primary fails, a new membership is emitted that determines the new primary. Since only one membership is active at a time, no two replicas can believe to be the primary simultaneously.

The replication protocol of uKharon-KV works as follows: The primary P replicates all clients' requests to a single backup B by RDMA-writing them to a dedicated buffer on the latter. In parallel, P *speculatively* executes the requests. Upon completion of the RDMA WRITE, the primary checks that the membership in which P is the primary is still active. If that is the case, P replies to the client. Otherwise, P drops the request. Upon membership change, B waits for the new membership—in which it is the primary—to become active. Then, B scans the local buffer that was dedicated to P and applies all unprocessed requests in it. Only then B starts pro-

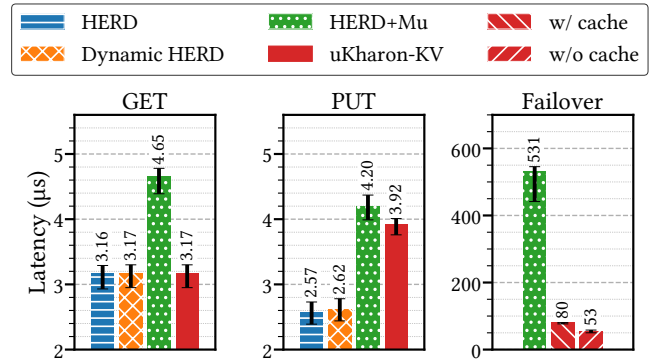


Figure 5: Latency comparison (left) of vanilla HERD, Dynamic HERD, HERD+Mu, uKharon-KV. Failover time comparison (right) of HERD+Mu and uKharon-KV. HERD+Mu uses 3-way replication; uKharon-KV uses its deadbeat. Bar height shows 95th %-ile latency; numerical label shows the 95th %-ile; error bars show the median and 99th %-ile.

cessing clients' requests. The client's failover time is the time interval between the client's last successful request to P and its first successful request to B (as the new primary).

If P 's speculative execution turns out to be incorrect, its state may diverge from the one of the new primary B . uKharon-KV, however, does not follow the common practice of rolling back unsuccessful speculations, because our prototype adopts a simple design: when a node is removed from the membership, it is not allowed to re-enter the system. Thus, the state of the old primary P is no longer used when B takes over, hence skipping the rollback.

Replication latency. We compare the latency of HERD, HERD+Mu and uKharon-KV. For HERD, we deploy a single node. For HERD+Mu, we deploy three nodes, a primary and two backups, all of which execute an instance of HERD and Mu. For uKharon-KV, we deploy a primary and a backup, both running uKharon-KV, as well as three coordinators. For these experiments, a HERD client connects to the primary and issues PUT and GET requests. We measure the time it takes for a client to complete a request and compute the median, the 95th and the 99th percentiles over 10 million requests.

Figure 5 shows the end-to-end latency of vanilla HERD and of both replication approaches. In vanilla HERD, PUTs are more efficient than GETs by 23%, due to the way HERD handles the two types of requests. Briefly, PUTs rely mostly on RDMA WRITES, which is the most efficient RDMA verb [25], while GETs rely mostly on RDMA SENDS. For reference, we also show the latency of Dynamic HERD, which uses uKharon's `Active` method in the critical path of executing clients' requests, as explained in section 7.1. We verify, once again, the efficiency of the `Active` method. At the 95th percentile, Dynamic HERD's requests are delayed by 10ns (for GETs) and 50ns (for PUTs), compared with vanilla HERD.

The two replicated solutions exhibit different costs.

HERD+Mu replicates all requests, regardless of whether they are PUTs or GETs, while uKharon-KV replicates only PUTs. HERD+Mu does not distinguish between PUTs and GETs, because in Mu the primary uses the result of replication (whether it is successful or not) to determine if it is still the primary or not. If Mu were to skip the replication of GETs, inconsistency would occur (see §2.1). On the other hand, uKharon-KV executes GETs locally, without replicating them, since the primary relies on the `Active` method to determine if its data is stale or not. Also, observe that uKharon-KV replicates PUTs approximately 300ns faster than Mu. This improvement is merely attributed to the speculative approach adopted by uKharon-KV. In HERD+Mu, the primary executes the request *after* it has been replicated to a majority. On the other hand, the primary in uKharon-KV executes the request in *parallel* to the replication to the backup. Thus, our solution hides the cost of executing the request, which is approximately 300ns, as shown by the difference of the two rightmost bars in the middle plot of Fig. 5. Regardless, uKharon-KV provides the same fault tolerance as Mu, even with one less replica: if a single replica crashes in either HERD+Mu or uKharon-KV, the system remains operational but cannot tolerate another failure. Fundamentally, both HERD+Mu and uKharon-KV assume a majority of correct nodes, the former among the replicas and the latter among the coordinators.

Failover. We compare the failover latency of uKharon-KV with HERD+Mu in the event of userspace failures. We run uKharon-KV in two configurations. In the first one, clients directly RDMA-read from coordinators to renew their lease. In the second one, clients go to lease caches. The third graph of Figure 5 shows that HERD+Mu has a 95th-percentile failover time of 531 μ s. This number is almost half of what Mu’s authors report since we fine-tuned their failure detector for our own setup. At the same time, uKharon-KV without cache (resp. with) achieves a 10 \times improvement (resp. 6.5 \times) at 53 μ s (resp. 80 μ s) of end-to-end failover time.

8 Related Work

Membership services in general. They are widely used in the data center. Distributed data processing apps (e.g., Kafka [30], MapReduce [10]), storage systems (e.g., Cassandra [31], HDFS [46]) and orchestration tools (e.g., Mesos [18]) rely on Zookeeper [20] for leader election, membership management, locks, watches, etc. uKharon focuses on membership management, yet it can be extended to support Zookeeper’s features. Indeed, uKharon-KV (excluding the lack of durability) offers similar guarantees to the strongly consistent KV-store of Zookeeper, which comprises its basic building block. ZooKeeper’s strongly consistent KV-store that forms its basis. For instance, locks can be implemented on top of uKharon-KV by extending its interface with `CompareAndSwap`. Watches, being an unreplicated pub/sub sys-

tem, only require modifying uKharon-KV’s primary. The important difference is that Zookeeper is not suitable for the microsecond scale and does not exploit RDMA.

Failure detection in the data center. A common approach to detect failures is to use end-to-end timeouts, which are hard to set. Falcon [35] proposes to use inside information in order to build faster and more accurate failure detectors by relying on hierarchies of specialized detectors. It maximizes accuracy by killing suspected processes. Albatross [34] is slightly more forgiving and isolates suspected processes so that they cannot affect the state of the system. Pigeon [33] provides fine-grained reports that end applications use to act accordingly. We embrace Falcon’s philosophy and use RDMA-tailored failure detectors to operate at the microsecond scale.

Time-bound leases. Time-bound leases are widely used to implement consistent distributed applications at the price of some synchrony assumptions. They are often provided by a distributed coordination framework such as ZooKeeper [20] or etcd [14]. Leases are used for leader election [48], as well as for guarding memberships (e.g., in FaRM [12] and Hermes [26]). uKharon guards memberships with purely client-side leases. As a result, uKharon brings leases down to a few tens of microseconds and only assumes bounded clock drift instead of loosely synchronized clocks as in Hermes.

9 Conclusion

Continuous breakthroughs in data center fabrics have paved the way for microsecond applications. A key challenge for building tail-tolerant software at scale is for applications to react fast to events such as reconfigurations and failures. Yet, existing microsecond applications lack an equally fast membership service to provide microsecond dynamicity. This lack is counter-intuitive, as the vast ecosystem built around ZooKeeper showcases the usefulness of membership services. uKharon fills this gap by being the first membership service tailored to microsecond scale applications. To achieve this demanding target, uKharon relies on (1) a multi-level failure detector, (2) a consensus engine that takes advantage of RDMA CAS, as well as (3) leases, all of which have been carefully designed to operate in the microsecond envelope. We used uKharon to implement uKharon-KV, a replicated KV-cache which outperforms the state of the art in latency while improving its failover time by up to 10 \times .

Acknowledgments

We thank our NSDI ’22 and ATC ’22 anonymous reviewers as well as our shepherd, Abhinav Duggal, for their valuable comments. We would also like to thank our anonymous artifact evaluators for reviewing our implementation. This work was partly funded by Huawei Technologies.

References

- [1] Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 51–60, July 2018.
- [2] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 599–616, 2020.
- [3] Marcos K. Aguilera and Michael Walfish. No time for asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems, HotOS’09*, page 3, USA, 2009. USENIX Association.
- [4] Motti Beck and Michael Kagan. Performance evaluation of the RDMA over ethernet (RoCE) standard in enterprise data centers infrastructure. In *Proceedings of the 3rd Workshop on Data Center-Converged and Virtual Ethernet Switching*, pages 9–15, 2011.
- [5] Christian Cachin, Rachid Guerraoui, and Lu s Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.
- [6] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.
- [7] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, July 1996.
- [8] Intel Corporation. Volume 3B: System Programming Guide, Part 2. In *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Intel Corporation, 2016.
- [9] Jeffrey Dean and Luiz Andr  Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, feb 2013.
- [10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [11] Travis Downs. A benchmark for low-level CPU micro-architectural features. <https://github.com/travisdowns/uarch-bench>. Accessed 2022-05-25.
- [12] Aleksandar Dragojevi , Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, April 2014.
- [13] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.
- [14] Etcd. <https://etcd.io>. Accessed 2022-05-25.
- [15] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smart-nics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.
- [16] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.
- [17] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, April 1985.
- [18] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.
- [19] Peter H Hochschild, Paul Turner, Jeffrey C Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. Cores that don’t count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 9–16, 2021.
- [20] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference (ATC)*, June 2010.
- [21] Zsolt Istv n, David Sidler, and Gustavo Alonso. Caribou: Intelligent distributed storage. *Proceedings of the VLDB Endowment*, 10(11):1202–1213, 2017.
- [22] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soul , Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-RTT coordination. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 35–49, April 2018.
- [23] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *USENIX*

Symposium on Networked Systems Design and Implementation (NSDI), pages 1–16, February 2019.

- [24] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *ACM Conference on SIGCOMM*, pages 295–306, August 2014.
- [25] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance RDMA systems. In *USENIX Annual Technical Conference (ATC)*, pages 437–450, June 2016.
- [26] Antonios Katsarakis, Vasilis Avrielatos, M R Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojević, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 201–217, March 2020.
- [27] Colin King. stress-ng: A tool to load and stress a computer system. <https://github.com/ColinIanKing/stress-ng>. Accessed 2022-05-25.
- [28] Patrick Knebel, Dan Berkram, Al Davis, Darel Emmot, Paolo Faraboschi, and Gary Gostin. Gen-z chipset for exascale fabrics. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–22. IEEE Computer Society, 2019.
- [29] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *European Conference on Computer Systems (EuroSys)*, April 2020.
- [30] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.
- [31] Avinash Lakshman and Prashant Malik. Cassandra—a decentralized structured storage system. In *International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, October 2009.
- [32] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.
- [33] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2013.
- [34] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming uncertainty in distributed systems with help from the network. In *European Conference on Computer Systems (EuroSys)*, April 2015.
- [35] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the FALCON spy network. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2011.
- [36] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM '19*, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.
- [37] Linux Kernel Developers. NO_HZ: Reducing Scheduling-Clock Ticks. https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt. Accessed 2022-05-25.
- [38] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, 2013.
- [39] René Peinl, Florian Holzschuher, and Florian Pfitzer. Docker cluster management for the cloud-survey results and own solution. *Journal of Grid Computing*, 14(2):265–282, 2016.
- [40] Gregory F Pfister. An introduction to the InfiniBand architecture. *High performance mass storage and parallel I/O*, 42(617-632):10, 2001.
- [41] Marius Poke and Torsten Hoefler. DARE: High-performance state machine replication on RDMA networks. In *Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 107–118. ACM, June 2015.
- [42] Linux RDMA. perfctest: Infiniband verbs performance tests. <https://github.com/linux-rdma/perfctest>. Accessed 2022-05-25.
- [43] Red Hat, Inc. RHEL for Real Time Timestamping. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/chap-timestamping. Accessed 2022-05-25.
- [44] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.
- [45] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. Ieee, 2010.

- [46] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. Ieee, 2010.
- [47] Swaminathan Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730, 2012.
- [48] Anish Sukumaran and Vincent Gerard Nicotra. Lease based leader election system, May 29 2018. US Patent 9984140.
- [49] Mellanox Technologies. RDMA aware networks programming user manual. rev 1.7. <https://docs.nvidia.com/networking/spaces/viewspace.action?key=RDMAAwareProgrammingv17>. Accessed 2022-05-25.
- [50] Stephen Van Doren. HOTI 2019: Compute Express Link. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 18–18. IEEE, 2019.
- [51] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS: Fast and scalable paxos on RDMA. In *Symposium on Cloud Computing (SoCC)*, pages 94–107, September 2017.
- [52] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation OSDI 18*, pages 233–251, 2018.
- [53] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–104, October 2015.
- [54] Tian Yang, Robert Gifford, Andreas Haeberlen, and Linh Thi Xuan Phan. The synchronous data center. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 142–148, 2019.
- [55] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 10(6):685–696, February 2017.

A One-sided Paxos

A.1 Assumptions

In the next subsections, we consider the M&M model [1]. It allows processes to both pass messages and share memory.

We assume that communication channels are lossless and have FIFO semantics, which is ensured by InfiniBand’s Reliable Connections. The system has n processes $\Pi = \{p_1, \dots, p_n\}$ that can attain the roles of *proposer* or *acceptor*. There are p proposers and n acceptors. Up to $p - 1$ proposers and $\lfloor \frac{n-1}{2} \rfloor$ acceptors may fail by crashing. As long as a process is alive, its memory is remotely accessible. When a process crashes, subsequent operations to its memory hang forever. We assume partial synchrony for consensus’s liveness [17].

A.2 One-sided RPC

In this section, we prove that the one-sided RPCs of Algorithm 1 are equivalent to two-sided RPCs when not obstructed. Moreover, we prove that when equivalence is violated (due to obstruction), one-sided RPCs have no side effects. We assume that both `compare` and `f` are deterministic.

Lemma A.1. *If `cas-rpc` does not abort, `rpc` and `cas-rpc` are equivalent.*

Proof. An execution of `rpc` solely depends on the value of `state` and the input value x . We denote such execution of `rpc` as $\langle state, x \rangle_{rpc}$. If an execution of `cas-rpc` does not abort, it solely depends on the value of `expected` fetched at line 2 and the input value x . We denote such execution of `cas-rpc` as $\langle expected, x \rangle_{cas-rpc}$.

We show that any execution $\langle s, x \rangle_{rpc}$ is equivalent to the execution $\langle s, x \rangle_{cas-rpc}$ in the sense that both `rpc` and `cas-rpc` will have the same `state` value and return the same projection at the end of their execution.

If an execution $\langle s_1, x \rangle_{rpc}$ makes the comparison at line 3 fail, then `state` is not modified and `proj(s1)` is returned. In the execution $\langle s_1, x \rangle_{cas-rpc}$, the comparison at line 3 will also fail and `proj(s1)` is also returned without modifying the remote state. In this case, both executions are equivalent.

If an execution $\langle s_2, x \rangle_{rpc}$ makes the comparison at line 3 succeed, then `state` is modified to `f(s2, x)` and `proj(f(s2, x))` is returned. In the execution $\langle s_2, x \rangle_{cas-rpc}$, the comparison at line 3 will also succeed. As the execution is assumed not to abort, the CAS will succeed. Thus the remote state will atomically be updated from s_2 to `f(s2, x)` and `proj(f(s2, x))` is also returned. In this case, both executions are also equivalent. \square

Lemma A.2. *If `cas-rpc` aborts, it has no side effects.*

Proof. If `cas-rpc` aborts, the comparison at line 7 has failed. This implies that the CAS failed and thus that `state` is unaffected by the execution. \square

From lemmas A.1 and A.2, `cas-rpc` exhibits all-or-nothing atomicity. We now prove that such a transformation is obstruction-free.

Lemma A.3. *If `cas-rpc` runs alone, it does not abort.*

Proof. Let's assume by contradiction that `cas-rpc` runs alone and aborts. For `cas-rpc` to abort, the comparison at line 7 must have failed. This implies that the CAS at line 6 failed due to state not matching expected. `state` must thus have been updated between lines 2 and 6. This implies a concurrent execution, hence a contradiction. \square

A.3 Consensus and Abortable Consensus

In the consensus problem, processes *propose* individual values and eventually irrevocably *decide* on one of them. Formally, consensus has the following properties:

Termination Every correct process eventually decides once.

Uniform agreement If v and v' are decided on, then $v = v'$.

Validity If v is decided on, v is the input of some process.

We implement consensus by composing two abstractions:

- *Abortable consensus* [5], an abstraction weaker than consensus that is solvable in the asynchronous model,
- *Eventually perfect leader election* [7], the weakest failure detector required to solve consensus.

Abortable consensus is identical to consensus except for:

Termination Every correct process eventually decides once or aborts.

Decision If a single process proposes infinitely many times, it eventually decides.

A.4 One-sided Abortable Consensus

Algorithm 3 appears in [5] and implements abortable consensus. Algorithm 4 transforms algorithm 3 by replacing its RPCs with CAS-based RPCs. This transformation causes it to abort strictly more than the original algorithm. To see why, consider the following execution: Let proposers P_1 and P_2 concurrently initiate the Prepare phase with respective proposals 1 and 2. Both fetch the remote state and get $\langle 0, 0, \perp \rangle$. Then, P_1 succeeds in writing its proposal to acceptor A_1 . Later on, the CAS of P_2 fails at A_1 as the value is now $\langle 1, 0, \perp \rangle$ instead of the expected $\langle 0, 0, \perp \rangle$. Thus, P_2 aborts even if it had a larger proposal number than P_1 . The more relaxed comparison in the original algorithm would not have caused P_2 to abort.

Lemma A.4. *Algorithm 4 preserves Decision.*

Proof. If a single process proposes infinitely many times, it will eventually run the one-sided RPCs obstruction-free. By Lemma A.3, this guarantees that the one-sided RPCs will eventually terminate without aborting. In such case, Lemma A.1 guarantees the execution to be equivalent to one of the original algorithm. Thus, the transformation preserves the decision property of Algorithm 3. \square

Algorithm 3: Paxos's Abortable Core

```

1  Proposers execute:
2  decided = False
3  proposal = id
4  proposed_value =  $\perp$ 

6  def propose(value):
7      proposed_value = value
8      prepare()
9      accept()

11 def prepare():
12     proposal = proposal + | $\Pi$ |
13     broadcast (Prepare | proposal)
14     wait for a majority of (Prepared | ack, ap, av)
15     adopt av with highest ap as proposed_value
16     if any not ack: abort

18 def accept():
19     broadcast (Accept | proposal, proposed_value)
20     wait for a majority of (Accepted | mp)
21     if any mp > proposal: abort
22     trigger once (Decide | proposed_value)

24 Acceptors execute:
25 min_proposal = 0
26 accepted_proposal = 0
27 accepted_value =  $\perp$ 

29 upon (Prepare | proposal):
30     if proposal > min_proposal: min_proposal = n
31     reply (Prepared | min_proposal == n, accepted_proposal,
            $\hookrightarrow$  accepted_value)

33 upon (Accept | proposal, value):
34     if proposal  $\geq$  min_proposal:
35         accepted_proposal = min_proposal = n
36         accepted_value = value
37     reply (Accepted | min_proposal)

```

Lemma A.5. *Algorithm 4 preserves Termination.*

Proof. Assuming a majority of correct acceptors, CASes will eventually complete at a majority. Due to the absence of loops or blocking operations inside `prepare`, `accept`, `cas_prepare` and `cas_accept` in algorithm 4 (apart from waiting for the completion of CASes at a majority), a proposer that invokes `propose` will either abort or decide. \square

Algorithms 3 and 4 differ only in some executions where the transformed algorithm aborts whereas the original does not. Nevertheless, aborting does not violate safety, as we show next.

Lemma A.6. *Algorithm 4 preserves the safety properties.*

Proof. Assume, by contradiction, that adding superfluous abortions in Algorithm 3 violates safety. Consider an execution E_1 , where processes $\{P_1, \dots, P_n\}$ deviate from the algorithm and abort at times $\{t_1, \dots, t_n\}$ after which the global state is $\{S_1, \dots, S_n\}$ and safety is violated. Also, consider another execution E_2 , where processes $\{P_1, \dots, P_n\}$ crash at times $\{t_1, \dots, t_n\}$ after which the global state is $\{S_1, \dots, S_n\}$. In execution E_1 , safety is violated. On the other hand, execution E_2 preserves safety, since Algorithm 3 tolerates arbitrarily many proposer crashing. The two executions, however, are

Algorithm 4: One-sided Abortable Consensus

```

1  Acceptors execute:
2  state = { min_proposal: 0, accepted_proposal: 0,
           ↪ accepted_value: ⊥}

4  Proposers execute:
5  proposal = id
6  proposed_value = ⊥

8  def propose(value):
9    proposed_value = value
10   prepare()
11   accept()

13 def prepare():
14   proposal = proposal + |Π|
15   async cas_prepare(p) for p in Acceptors
16   wait for a majority to return ⟨ack, ap, av⟩
17   if any not ack: abort
18   adopt av with highest ap as proposed_value

20 def accept():
21   async cas_accept(p) for p in Acceptors
22   wait for a majority to return mp
23   if any mp > proposal: abort
24   trigger once (Decide | proposed_value)

26 def cas_prepare(p):
27   expected = fetch_state(p)
28   if not proposal > expected.min_proposal:
29     return (False, expected.accepted_proposal, expected,
           ↪ accepted_value)
30   move_to = expected
31   move_to.min_proposal = proposal
32   read = statep.cas(expected, move_to)
33   if read == expected:
34     return (True, expected.accepted_proposal, expected,
           ↪ accepted_value)
35   abort

37 def cas_accept(p):
38   expected = fetch_state(p)
39   if not proposal ≥ expected.min_proposal:
40     return expected.min_proposal
41   move_to = expected
42   move_to.min_proposal = proposal
43   move_to.accepted_proposal = proposal
44   move_to.accepted_value = proposed_value
45   read = statep.cas(expected, move_to)
46   if read == expected:
47     return expected.min_proposal
48   abort

```

indistinguishable, hence a contradiction. Thus, Algorithm 4 preserves safety regardless of how often it aborts. \square

Theorem A.7. *Algorithm 4 implements abortable consensus.*

Proof. The result follows directly by composing lemmas A.4, A.5 and A.6. \square

A.5 Streamlined One-sided Algorithm

In this section, we make Algorithm 4 efficient in order to increase its practicality.

First, it is not required to fetch the remote state at the start of each RPC. As it is safe to have stale `expected` states, it is safe to use states deduced from previous CASes. Predicted states can thus be initialized to $\langle 0, 0, \perp \rangle$ and updated each time a CAS completes (either succeeding or not). Moreover,

wrongly predicting states can only result in superfluous aborts which have been proven to be safe by Lemma A.6. Thus, it is safe to optimistically assume that onflight CASes will succeed. Second, in the Prepare phase, the `proposal` variable can be increased upfront to value higher than any predicted remote `min_proposal` to reduce predictable aborts.

Algorithm 5: Streamlined One-sided Abortable Consensus

```

1  Acceptors execute:
2  state = { min_proposal: 0, accepted_proposal: 0,
           ↪ accepted_value: ⊥}

4  Proposers execute:
5  predicted[] = { 0, 0, ⊥}
6  proposal = id
7  proposed_value = ⊥

9  def propose(value):
10   proposed_value = value
11   prepare()
12   accept()

14 def prepare():
15   while any predicted[.].min_proposal ≥ proposal:
16     proposal = proposal + |Π|
17   for p in Acceptors:
18     move_to[p] = {min_proposal: proposal, ..predicted[p]}
19     reads[p] = async statep.cas(predicted[p], move_to[p])
20   wait until majority of states are read
21   for p in Acceptors:
22     if reads[p] ∈ {predicted[p], ⊥}:
23       predicted[p] = move_to[p]
24     else:
25       predicted[p] = reads[p]
26   if any CAS failed: abort
27   adopt proposed_value from predicted accepted_values with
           ↪ highest accepted_proposal if any

29 def accept():
30   reads = ⊥|Acceptors|
31   move_to = (proposal, proposal, proposed_value)
32   for p in Acceptors:
33     reads[p] = async statep.cas(predicted[p], move_to)
34   wait until majority of states are read
35   if any CAS failed:
36     for p in Acceptors:
37       if reads[p] ∈ {predicted[p], ⊥}:
38         predicted[p] = move_to
39     else:
40       predicted[p] = reads[p]
41   abort
42   trigger once (Decide | proposed_value)

```

With the aforementioned optimisations, Algorithm 4 is transformed into Algorithm 5. Notably, the liveness of the resulting algorithm is preserved: Let's assume that a *single* proposer runs infinitely many times. Eventually, it will run obstruction-free. In the worst case, each time it will abort at line 26 or 41 because of a single wrong guess and update its prediction. The optimistic update of expected states at lines 23 and 38 and the FIFO semantics of communication links provide that, once a remote state is correctly guessed, any later CAS will succeed. Thus, after at most n runs, all CASes will succeed and the proposer will decide.

A.6 Overcoming Limited CAS Size

As explained in Section 5.3.1, the RDMA hardware limits the size of CASes. Thus, proposal fields will overflow after 2^{16} attempts. In such an unlikely scenario, our consensus engine falls back to traditional RPC: Once the RDMA-exposed `min_proposal` of an acceptor reaches $2^{16} - |\Pi|$, proposers switch to RPC to communicate with this specific acceptor. Acceptors check `state` and, if it is above the threshold, initiate the standard RPC version of Paxos with the `min_proposal`, `accepted_proposal` and `accepted_value` variables initialized to match `state`.

B Active Method Correctness

In this section, we provide a formal definition and a proof of correctness of the `Active` method described in Section 6.

B.1 Formal Definition

`Active(Membership) → bool` has the following properties:

Monotonicity If `Active(M')` returns `true` at any process, future calls `Active(M)` with $M < M'$ will return `false`.

Convergence If M is the last membership to be decided (if any), invoking `Active(M)` will eventually return `true` at all correct processes.

Definition 1. If `Active(M)` returns `true`, then M is considered active at the linearization point of the call.

Definition 2. If M is active at times t and t' , then it is considered active in the interval $[t, t']$.

From these simple properties and definitions, it follows that no two active memberships can overlap.

Theorem B.1. Only one membership can be active at a time.

Proof. Assume by contradiction that M and M' ($M < M'$) are simultaneously active. By definition, `Active(M)` must have returned `true` after `Active(M')` returned `true`. This breaks Monotonicity, hence a contradiction. \square

B.2 Non-leased Active Membership

We prove the correctness of uKharon's implementation of `Active`. We assume no gaps in the sequence of decided memberships. This is enforced by coordinators by not proposing the $(k+1)$ -th membership until the k -th is decided.

Lemma B.2. Algorithm 6 ensures Monotonicity.

Proof. `Active` can only be called on decided memberships. Let M and M' be two decided memberships with $M < M'$. If `Active(M')` returned `true`, by the no-gap assumption, all

Algorithm 6: Active built on top of the consensus engine

```
1 def Active(M) → bool:
2   reads =  $\perp_{|\text{Acceptors}|}$ 
3   for p in Acceptors:
4     reads[p] = async paxos[M.id + 1].slotp.read()
5   wait until majority of slots are read
6   if all slots are not accepted:
7     return true
8   propose_membership(M.id + 1, first accepted value)
9   return false
```

memberships between M and M' have been decided. Because M' 's successor has been decided, a majority of acceptors' slots $M.id + 1$ have been written. Thus, `Active(M)` will read at least one non-empty slot and return `false`. \square

Lemma B.3. Algorithm 6 ensures Convergence.

Proof. Assume by contradiction that M is the last decided membership and `Active(M)` never returns `true` at some correct process. Thus, this process executes line 8, which means that it proposes a new membership. Given that the process is correct, some membership with $id\ M.id + 1$ will eventually be decided. Therefore, M is not the last membership, hence a contradiction. \square

Theorem B.4. Algorithm 6 implements `Active`.

Proof. Follows directly from Lemmas B.2 and B.3. \square

B.3 Leased Active Membership

Algorithm 2 reduces communication by leasing the output of Algorithm 6. We prove that it preserves `Active`'s properties.

Lemma B.5. Algorithm 2 preserves Monotonicity.

Proof. Let e be an execution of `Active(M)` that returned `true`. e either returned at line 9 or at line 12 with $t > t_{start}$. We denote the former case *leased(M)* and the latter *checked(M)*. Assume by contradiction that `Active(M')` returned `true` in an execution e_1 and then `Active(M)` returned `true` in an execution e_2 with $M < M'$. Either:

- *leased(M)*: In e_2 , `majority_active(M)` returned `true` at most δ before `Active(M)` returned `true`. In e_1 , lines 5–7 ensure that M' was decided at least δ before `Active(M')` returned `true`. Thus, `majority_active(M)` returned `true` after M' was decided. However, because M' has been decided, a majority of acceptors' slots $M'.id = M.id + 1$ must have been written. Thus, `majority_active(M)` should have read at least one non-empty slot and returned `false`. Hence, a contradiction.
- *checked(M)*: `majority_active(M)` returned `true` after `majority_active(M')` returned `true`. This breaks `majority_active`'s Monotonicity, hence a contradiction.

□

Lemma B.6. *Algorithm 2 preserves Convergence.*

Proof. Assume that M is the last membership to be decided. Thus, `majority_active(M)` will eventually always return `true`. At most δ after `Active(M)` returns for the first time, t_{start} will be in the past and `leased_membership` set to M . Thus, eventually, the `else` branch at line 8 will always be visited and either return `true` via line 9 or 12. □

Theorem B.7. *Algorithm 2 implements Active.*

Proof. Follows directly from Lemmas B.5 and B.6. □

C Clocks

uKharon relies on hardware timestamps to check if a membership is `Active`. When using modern Intel processors, Linux has three available clocksources: `tsc`, `hpet` and `acpi_pm`. The `tsc` clocksource is the most efficient and requires 20-25ns to take a timestamp [11].

Architectural considerations. The `tsc` clocksource uses Intel's TSC hardware to measure time accurately. TSC stores the number of cycles executed by the CPU after the latest reset. Traditionally, TSC is considered an unreliable way to take timestamps. The reason is that Intel processors have variable clock speed, thus the number of cycles does not correspond to wallclock time. However, modern Intel processors have three features [8]: *Constant TSC*, *Nonstop TSC* and *Invariant TSC* which solve this problem. The combination of these features results in a TSC that is incremented at a constant rate regardless of the power state of the processor. As a result, it is safe to use this counter for efficient timestamping.

TSC synchrony. In Intel processors, every core has its own TSC. All processors in the same socket start the TSC hardware using the same `RESET` signal, thus the absolute values of the TSC across cores of the same socket match. This means that one can compare safely the values of TSC across different cores, assuming that all TSCs run at the same frequency. Because this assumption does not always hold, Linux determines the base frequency of every core during boot and uses this frequency to convert clock cycles to wallclock time. To accomplish it, Linux uses the more accurate (and more expensive) `hpet`.

uKharon takes further care to deal with TSC synchrony. More precisely, it checks for the synchronization of TSC between cores using a ping-pong test. In this test, core A takes a timestamp t_1 and signals core B to do the same. Core A signals core B by writing to a lock-free Single-Producer Single-Consumer (SPSC) queue that is polled by B. When B receives the signal it also takes a timestamp t_2 and sends it back to A (using another SPSC queue). Upon reception of the timestamp from B, core A takes the last timestamp t_3 . In

our test we confirm that always $t_1 < t_2 < t_3$. Additionally, in our hardware, the minimum difference between t_1 and t_2 is $\epsilon = \min(t_2 - t_1)$ is 64ns. uKharon takes ϵ into consideration by incorporating into the leases as follows: Suppose a lease is valid for a duration of δ starting at time t . uKharon considers that the lease starts at time $t + \epsilon$ and has a duration of $t + d - 2\epsilon$.

Inter-machine clock drift. In order to ensure that active memberships do not overlap, uKharon assumes that clock drift is bounded, i.e., that time passes approximately at the same speed on different machines. This assumption is necessary to enable client-side leases. It guarantees that after a lease duration period, leases across all clients will have to be renewed. Our system is built to tolerate clock drift, as long as this drift is bounded. We experimentally determine an upper bound for the clock drift with a simple test. In this test, machine A takes a timestamp t_1 and pings machine B to wait for 1 minute before replying back to it. Upon reception of B's response, A takes another timestamp t_2 . It then computes $t_2 - t_1$ and compares it to the expected 1 minute measured by B (after removing the communication delay). We repeat this test several times and determine that the clock drift between machines differs by at most 0.001%. uKharon incorporates inter-machine clock drift by waiting $1.01 \times \delta$ upon membership discovery, ensuring that when leases become active on a new membership, everyone's leases on the previous membership will have expired.

D Artifact

Abstract

The evaluated artifact is provided as a git repository and contains the source code of uKharon, build instructions and deployment scripts to run the experiments presented in this paper.

Scope

The artifact contains code and steps to reproduce results obtained in Figure 3, Figure 4, Figure 5 and Table 2.

Contents

The artifact contains the source code of uKharon, including the custom kernel modules. It also contains the patches to create the custom Linux kernel, as well as the patches required for HERD [24] and Mu [2]. The artifact describes how to build everything presented in the paper, including the custom Linux kernel and the solutions we compare against. It also describes how to deploy the built binaries.

Hosting

The artifact source code for uKharon is available at <https://github.com/LPD-EPFL/ukharon>. All the necessary instructions are provided in the README.md file.

Requirements

Building uKharon requires an x86-64 system set-up with Ubuntu 20.04 LTS. Executing uKharon requires 8 machines equipped with Ubuntu 20.04 LTS, RDMA over InfiniBand, ability to install a custom kernel and custom kernel modules, as well as ability to configure and use InfiniBand multicast groups.

KRCORE: A Microsecond-scale RDMA Control Plane for Elastic Computing

Xingda Wei^{1,2}, Fangming Lu¹, Rong Chen^{*1,2}, and Haibo Chen¹

¹Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University

²Shanghai AI Laboratory

Abstract

We present KRCORE, an RDMA library with a microsecond-scale control plane on commodity RDMA hardware for elastic computing. KRCORE can establish a full-fledged RDMA connection within $10\mu s$ (hundreds or thousands of times faster than verbs), while only maintaining a (small) fixed-sized connection metadata at each node, regardless of the cluster scale. The key ideas include virtualizing pre-initialized kernel-space RDMA connections instead of creating one from scratch, and retrofitting advanced RDMA dynamic connected transport with static transport for both low connection overhead and high networking speed. Under load spikes, KRCORE can shorten the worker bootstrap time of an existing disaggregated key-value store (namely RACE Hashing) by 83%. In serverless computing (namely Fn), KRCORE can also reduce the latency for transferring data through RDMA by 99%.

1 Introduction

The desire for high resource utilization has led to the development of elastic applications such as disaggregated storage systems [52, 16, 67]. Elasticity provides a quick increase or decrease of computing resources (e.g., processors or containers) based on application demands. Since the resources are dynamically launched and destroyed, minimizing the control path overheads—including process startup and creating network connections—is vital to applications, especially those with ephemeral execution time. Elastic applications typically have networking requirements. For instance, computing nodes in a disaggregated storage system access the data stored at the storage nodes across the network.

RDMA is a fast networking feature widely adopted in datacenters [53, 19, 13]. Unfortunately, RDMA has a slow control path: the latency of creating an RDMA connection (15.7ms) is 15,700X higher than its data path operation (see Figure 1(b)). As the latency of typical RDMA-enabled applications that require elasticity has reached to microsecond-scale (see Figure 1(a)), this high connection time may significantly decrease the application efficiency, e.g., increasing latency when expanding resources to handle load spikes. The cost is challenging to reduce because it not only includes software data structure initialization costs but also involves extensive hardware resource configurations, as RDMA offloads network processing to the network card (§2.3.1).

*Rong Chen is the corresponding author (rongchen@sjtu.edu.cn)

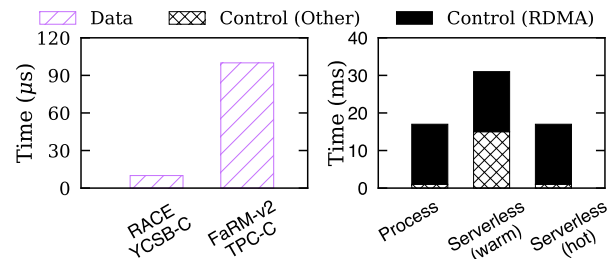


Fig. 1. (a) The execution time (**Data**) of typical elastic RDMA-enabled applications, and (b) the breakdown of control path costs. RACE [67] is a disaggregated key-value store. FaRM-v2 [46] is a database that can accelerate serverless transactions [63]. YCSB-C [11] and TPC-C [50] are representative benchmarks for each system. The serverless platform evaluated is Fn [43].

A common approach to avoiding the control path cost is to cache connections and share them with different applications. However, user-space RDMA connections can not be directly shared by different applications, because each app has its own exclusive driver data structure and dedicated hardware resources. Nevertheless, sharing a kernel-space RDMA connection is possible since applications share the same kernel (LITE [53]). However, LITE has performance and resource inefficiency issues (§2.3.2) in elastic computing, because it doesn't target this scenario. First, it still pays the initialization cost under cache misses. Second, caching all RDMA connections to all nodes is resource inefficient (e.g., taking several GBs of memory), especially when a production RDMA-capable cluster has reached a scale of more than 10,000 nodes [34]. Finally, sharing RDMA connections complicates the preservation of the low-level verbs interfaces, which is important to apply RDMA-aware optimizations [67, 55, 14, 57, 24, 25]. LITE only provides a high-level API.

We continue the line of reusing connections to boost the RDMA control path, and further overcome the issues mentioned above. We present **KRCORE**, a networking library with an ultra-fast control plane. KRCORE can establish a full-fledged RDMA-capable connection within $10\mu s$, only 0.05% and 0.22% of the verbs and LITE under cache misses, respectively. More importantly, KRCORE only needs a small amount of fixed-sized memory for the connection pool (e.g., 64MB), irrelevant to the cluster scale. Finally, KRCORE supports low-level RDMA interfaces compatible with existing RDMA-aware optimizations.

Supporting such a fast control plane seems to contradict our promise of a small fixed-sized connection pool. To achieve this, KRCORE makes a key innovation: we *retrofit* a less-studied yet widely supported advanced RDMA hardware feature—*dynamic connected transport* (DCT) [1]—to the kernel. DCT allows a single RDMA connection to communicate with different hosts. Its connection and re-connection are offloaded to the hardware and thus, are extremely fast (less than $1\mu s$). Our observation is that when virtualizing an established kernel-space DCT connection to different applications, they no longer pay the control path cost and memory consumption of ordinary RDMA connections.

In designing KRCORE, we found virtualizing DCT with a low-level API brings several new challenges, and we propose several techniques to address them (§3.1). First, DCT requires querying a piece of metadata to establish a new connection. Using RPC can not achieve a stable and low latency. Further, RPC needs extra CPU resources to handle DCT-related queries. Observing the small memory footprint of DCT metadata, we propose an architecture that deploys RDMA-based key-value stores to offload the metadata queries to one-sided RDMA READ (§4.2). Second, DCT has a lower data path performance than normal RDMA transport (RC) due to its dynamic connecting feature. The performance is mostly affected when a node keeps a long-term communication with another. Therefore, we introduce a hybrid connection pool that retains a few RC connections connected to frequently communicated nodes to improve the overall performance. KRCORE further adopts a transfer protocol that can transparently switch a virtualized connection from DCT to RC (§4.6). Finally, we propose algorithms to safely virtualize a shared physical QP to multiple applications with a low-level API (§4.4).

We implement KRCORE as a loadable Linux kernel module in Rust. We also extended an existing kernel-space RDMA driver (mlx-ofed-4.9) to bring DCT to the kernel. To the best of our knowledge, KRCORE is the first to achieve a microsecond-scale RDMA control plane. Although KRCORE is a general-purpose RDMA library, it really shines with elastic computing applications. Our experiments demonstrated that KRCORE can reduce the computing node startup time of a state-of-the-art production RDMA-enabled disaggregated key-value store (RACE [67]) by 83%, from 1.4s to 244ms (§5.3.1). For serverless computing—another popular elastic application, KRCORE can shorten the data transfer time over RDMA by 99%, from 33.3ms to $0.12\mu s$ (§5.3.2).

Our source code and experiments are available at <https://github.com/SJTU-IPADS/krcore-artifacts>.

2 Background and Motivation

2.1 The case for fast control path in elastic computing

KRCORE targets systems that require elasticity: the ability to automatically scale according to application demands. One such case is disaggregated storage systems where the computing nodes and storage nodes are separated and connected

by the network [52, 16, 67]. Under high loads, the system can dynamically add computing nodes for better performance: and they need to establish connections to the storage nodes on-the-fly. Another important case is serverless computing [22] where the platforms instantaneously launch short-lived tasks with containers¹. The launch time typically includes network connections [51].

Unlike long-running tasks (e.g., web servers), the control path (e.g., network creation) is typically on the critical path of elastic applications. For example, before executing the application code, a serverless function that issues database transactions must first establish network connections to remote storage nodes [63, 21]. With RDMA, the transaction latency has reached 10-100 μs [14, 57]. Reducing the control path costs—including launching a container and creating network connections—is therefore vital to the end-to-end execution time or tail latency of elastic applications (see Figure 1).

Much research has focused on reducing other control path costs, e.g., the container launch time to about 10ms [40] and even sub-millisecond [15]. However, only a few considered accelerating network connection creation [51], especially for RDMA. The control path of RDMA is indeed several orders of magnitude slower than its data path (e.g., 22ms vs. $2\mu s$ in §2.3). It is also orders of magnitude slower than the execution time of common elastic RDMA-enabled applications, or other control path costs (see Figure 1).

2.2 RDMA and queue pair (QP)

RDMA is a high bandwidth and low latency networking feature widely adopted in modern datacenters [53, 19]. It has two well-known primitives: *two-sided* provides a message passing primitive while *one-sided* provides a remote memory abstraction—the RDMA-capable network card (RNIC) can directly read/write server memory in a CPU-bypassing way.

Although RDMA is commonly used in the user-space, the kernel adopts the same *verbs* API (verbs), which exposes network connections as queue pairs (QPs). Each QP has a send queue (*sq*), a completion queue (*comp_queue*), and a receive queue (*recv_queue*). Both primitives follow a similar execution flow. To send a request (or a batch of requests), the CPU uses `post_send` to post it (or them) to the *send queue*. If the request is marked as *signaled*, the completion can be polled from the *completion queue* via `poll_cq`. For two-sided primitive, the CPU can further receive messages with `poll_cq` over the *receive queue*. Before receiving, one should use `post_recv` to post message buffers to the QP. Note that the CPU needs to register memory through `reg_mr` to give RNIC memory access permissions.

QP has several kinds of transport each with different capabilities. We focus on improving the control path performance of *reliable connected* QP (RCQP), as it is the most commonly used one that supports both RDMA primitives and is reliable.

¹Serverless platforms may use virtual machine (VM)s to run tasks, which is not the focus of our paper. §6 discusses how KRCORE can apply to VMs.

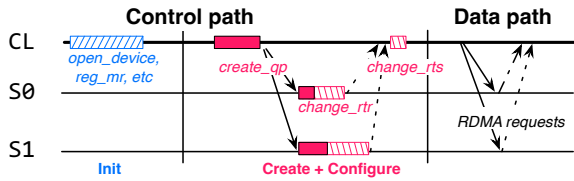


Fig. 2. The execution flow of a client (CL) communicating with two nodes (S0 and S1) using user-space verbs. `change_rtr` changes the QP to ready to receive status while `change_rts` changes the QP to ready to send status.

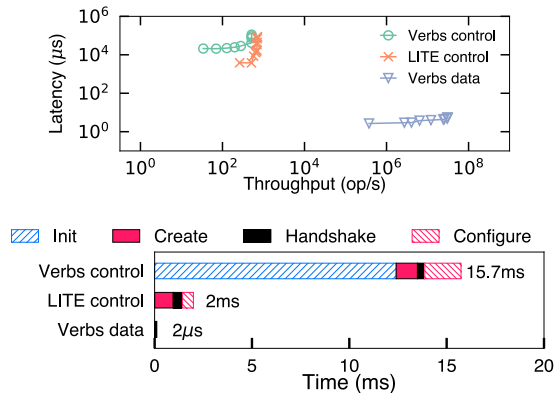


Fig. 3. (a) Huge performance gap btw. RDMA's control path and data path (issuing 8B READ) when connecting and communicating with one node. (b) A breakdown of RDMA control path time.

2.3 Analysis of RDMA control path costs

2.3.1 User-space control path costs

Consider the example in Figure 2 where a client sends RDMA requests to two nodes. The control path includes first initializing the driver context (**Init**)², creating the QPs (**Create**), exchanging the QP information to the remote peer with a **handshake** protocol and configuring the QPs to ready states (**Configure**). Figure 3(a) reports its latency, which is 7,850X higher than the data path (Verbs control vs. Verbs data).

Issue: High hardware setup cost. To quantify the costs in detail, Figure 3 breakdowns the control path time. We carefully optimize the connection handshake with RDMA's connectionless datagram [26], which is orders of magnitude faster than using TCP/UDP. Contradicting the common wisdom, exchanging the connection information through the network (**Handshake**) is **not** the dominant factor: **Handshake** only contributes 2.4% of the total time. The cost is dominated by communicating with the RNIC hardware for the connection setups. Consider the `create_qp` in **Create**: we found 87% of the `create_qp` time (361μs vs. 413μs) is waiting for the RNIC to create the hardware queues.

2.3.2 Existing kernel-space solution is insufficient

LITE [53] is the only kernel-space RDMA solution and is the closest to our work. It provides high-level remote memory *read*, *write* and *RPC* interfaces over the low-level verbs API (§2.2). LITE maintains an in-kernel connection pool that

²Including creating the protection domain and registering the memory.

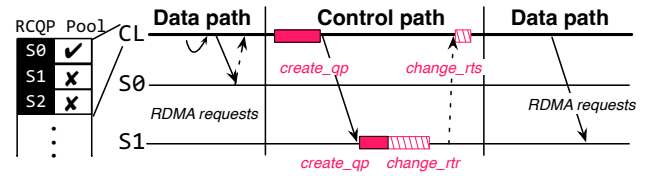


Fig. 4. The execution flow of a client (CL) communicating with two nodes (S0 and S1) with the kernel-space RDMA assuming that CL has cached a QP to S0 in its connection pool.

caches RCQPs connected to all nodes, which avoids the user-space **Init** (Figure 2) costs because applications share the same kernel-space driver data structures. However, it still has the following **issues** for elastic applications:

Issue#1: High cost connecting to a new node. If the RCQP of the target node is not cached, LITE must follow the same **Create** and **Configure** as user-space RDMA, e.g., S1 in Figure 4, which are non-trivial (2ms for each connection). Note that we have carefully optimized LITE's control path: LITE originally adopts a centralized cluster manager to create connections, which can only establish tens of QPs per second. We optimize it with a decentralized connection scheme using RDMA's connectionless datagram. The optimization achieves a 2ms per-connection latency and 712 QPs/second per node throughput (Figure 3), bottlenecked by the RNIC (see §2.3.1).

Issue#2: Huge memory consumption. Caching RCQPs connected to all other nodes can mitigate Issue#1. However, this strategy has huge per-machine memory consumption since the number of RCQPs needed scales linearly with the cluster size. In LITE, each QP consumes at least 159KB memory³, excluding the message buffers and receive queues (may share between different QPs via shared receive queue). Therefore, LITE would consume at least 1.52 GB memory per node for fast connection on a modern RDMA-capable cluster with more than 10,000 nodes[17].

Issue#3: Inflexible interface. LITE exposes a high-level RDMA API (e.g., a synchronous remote memory read), which simplifies sharing the same QP to different applications. However, it is inflexible to apply RDMA-aware optimizations widely adopted in the literature [67, 55, 14, 57, 24, 25], e.g., sending different read/write requests within a batch asynchronously. To utilize these optimizations, applications need verbs low-level API (§2.2). Unfortunately, directly executing the low-level API on a shared QP can easily corrupt the QP states (see §3.1), and interrupt application running. We carefully design the QP virtualization algorithms to correctly virtualize a shared QP with verbs's low-level API (§4.4).

³It configures the QP with 292 *sq* and 257 *comp_queue* entries, a common setup in RDMA-based systems. Each *sq* entry takes 448B while *cq* takes 64B. The driver would further round queues to fit the hardware granularity.

3 Approach and Overview

Opportunity: advanced RDMA transport (DCT). Dynamically Connected Transport (DCT) [1] is an advanced RDMA feature widely supported in commodity RNICs (e.g., from Mellanox Connect-IB [37] to ConnectX-7 [35]). DCT preserves the functionalities of RC and further supports dynamic connecting: a DCT QP (DCQP) can communicate to different nodes without user-initiated connections: RNIC can create DCT connections on-the-fly by piggybacking control plane messages with data plane ones. Since the connections are only processed in the hardware, DCT re-connection is extremely fast: our measured overhead is less than $1\mu s$. When using DC-QPs, the host only needs to specify the target node’s RDMA address and its DCT metadata (i.e., DCT number and DCT key) in each request.

Basic approach: virtualized kernel-space DCQP. The goal is to achieve an ultra-fast control plane for the applications. Our basic approach is to virtualize kernel-space DCQPs (as VQPs) to user-space applications. The observation is that DCT naturally addresses the costly creation overhead (**Issue#1**) and the huge memory consumption (**Issue#2**) of RCQPs (§2.3.2). A kernel-space solution further mitigates the user-space driver loading costs (§2.3.1).

VQP also supports low-level RDMA interfaces (e.g., `ibv_post_send`) with the necessary extended API suitable for elastic computing (§4.1). Therefore, users can flexibly apply existing RDMA-aware optimizations [24, 25, 57] (**Issue #3** in §2.3.2). Note that different VQPs can share the same physical QP in the kernel. Nevertheless, KRCORE provides an exclusively owned QP abstraction to the applications.

3.1 Challenges and solutions

#1. Efficient DCT metadata query. DCQP needs to query the DCT metadata before sending requests. Specifically, to allow communicating with DCT, the server must first create a *DCT target* identified by a key and number (DCT metadata). Afterward, the clients can piggyback the metadata in their requests to communicate with the created target.

A viable solution is to send an RPC to the target node to query the metadata using RDMA’s connectionless datagram (UD)⁴, which prevents control plane costs as UD is connectionless. However, it is inefficient in performance and CPU usage. First, the latency of RPC may vibrate to tens of milliseconds due to the scheduling and queuing overhead of the CPU. Second, KRCORE must deploy extra kernel threads to handle the queries.

Solution: RDMA-based meta server. We replicate the DCT metadata at a few global meta servers backed by RDMA-enabled key-value stores (KVS) [67, 58, 55, 13], meaning each node can query it with one-sided RDMA bypassing the CPU. To support one-sided RDMA while preventing QP over-

⁴It only supports two-sided RDMA.

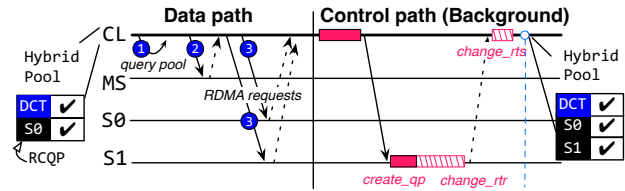


Fig. 5. The execution flow of a client (CL) communicating with two nodes (S0 and S1) with KRCORE. MS: meta server. Note that KRCORE always put the hardware control path (i.e., creating RC-QPs) in the background.

provisions, KRCORE only maintains a few RCQPs connected to nearby meta servers. Replicating the DCT metadata is practical because it is small: 12B is sufficient for one node to handle all requests from others.

#2. Performance issues of DCT. DCT is slower than RC in peak throughput and may incur high tail latency due to re-connection (§5.2). The performance is mostly affected when a node frequently sends requests to the same node.

Solution: virtualized hybrid QP. KRCORE manages a hybrid QP pool that stores both RC and DC QPs. A VQP can transparently switch between DC and RCQP (§4.6), allowing us to create RCQPs in the background on-the-fly without exposing the creations overhead to the applications.

#3. QP state protection. If we directly forward the VQP request (from `ibv_post_send`) from different applications to the (same) shared physical QP, QP’s physical states can easily be corrupted due to malformed requests or queue overflow, because verbs API assumes an exclusively owned QP. Bringing the QP back to a normal state is costly because it requires reconfiguration (the **Configure** in Figure 3 (b)).

Solution: pre-check. KRCORE carefully checks the physical queue capacity and request integrity before forwarding the requests to the physical QP. The overhead of these checks is negligible as they only involve simple calculations. Thus, we can avoid QP corruption while preserving the RDMA-aware optimizations (§4.4) of using low-level interfaces.

3.2 Execution flow and architecture

Execution flow. Applications can use KRCORE to create RDMA-capable connections in a few microseconds. Figure 5 presents its execution flow when communicating to two nodes. First, we find available RCQPs in the hybrid pool (1). If exists (S0), we directly virtualize it. Otherwise (S1), we choose a DCQP and fetch the target node’s DCT metadata (2) accordingly. Finally, we virtualize the selected QP so that the client can send RDMA requests with them (3).

To increase the likelihood of hitting RCQPs, KRCORE analyzes the host’s networking patterns and creates RCQPs in the background (e.g., to S1).

Architecture. Figure 6 presents the KRCORE library architecture. On each node, KRCORE is a loadable Linux

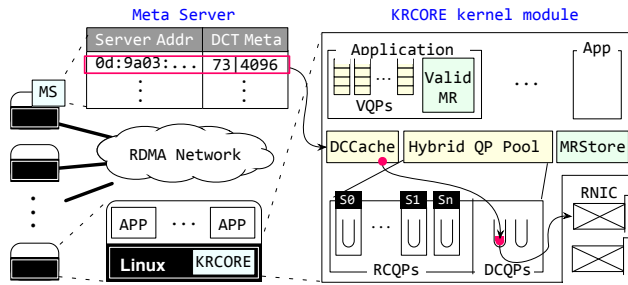


Fig. 6. An overview of KRCORE architecture.

```

KRCORE's extended verb's control path API
int qbind(ibv_qp *qp, ibv_gid gid, int port); ## like POSIX bind
int qconnect(ibv_qp *qp, ibv_gid gid, int port); ## like POSIX connect

KRCORE's extended verb's data path API
int qpop_msgs(ibv_qp *qp, int num_entries, ## like POSIX(accept) +
              ibv_qp **src_qp, ibv_wc *wc); ## verbs(ibv_poll_cq)

Example code: Client
1 ibv_qp_init_attr attr;
2 attr.qp_type = KRCORE_VQP;
3 qp = ibv_create_qp(..., &attr);
4 qconnect(qp, gid, port);
5 ibv_send_wr wr;
6 ibv_send_wr *bad_wr_ptr;
  ## send a message
7 ibv_post_send(qp, &wr, &bad_wr);

Example code: Server
1 ibv_qp_init_attr attr;
2 attr.qp_type = KRCORE_VQP;
3 qp = ibv_create_qp(..., &attr);
4 qbind(qp, gid, port);
5 ibv_qp *new_conn = NULL;
6 ibv_wc wc;
  ## receive a message
7 qpop_msg(qp, 1, &new_conn, &wc);

```

Fig. 7. The KRCORE extended API atop of verbs and a simplified use case. Lines in and are extended code for the client and server, respectively. Applications can also use the verb's data path call (e.g., `ibv_post_send`) to issue RDMA requests with KRCORE.

kernel module hosting per-application (e.g., VQP) and per-node (e.g., Hybrid QP Pool) data structures (§4.2). KRCORE also deploys meta servers (MS) on a few nodes to facilitate DCT metadata lookup. These servers are backed by DrTM-KV [58]—a state-of-the-art RDMA-enabled KVS—to accelerate the metadata lookup. The metadata is broadcasted by each machine during its boot time.

4 Detailed Design

4.1 Programming interface of KRCORE

To simplify application development and porting, it is important to keep backward compatibility between KRCORE and verbs, the de facto standard for using RDMA. In principle, KRCORE can provide the same interface with verbs similar to existing work (i.e., Freeflow [30]). However, verbs is not designed for elastic computing and may bring inflexibility or under-utilization of KRCORE. Therefore, we propose an extended API based on verbs inspired by Demikernel [64], as shown in Figure 7. Specifically, KRCORE introduces a new type of QP (VQP) with the following new primitives:

qconnect and qbind. The verbs API has no method for ‘connect’ commonly found in networking libraries. Therefore, developers have to implement and optimize RDMA connection setups themselves. We provide a `qconnect` API to abstract the fast connection provided by KRCORE. Specifically, after calling `qconnect` on a VQP to a remote host (identified by the RDMA address (gid) and a port), the VQP can issue one-sided and two-sided requests to it. Note that remote end must bind to the address using `qbind` beforehand so that the sender can issue two-sided requests, similar to POSIX `bind`.

qpops_msgs. RCQPs are one-to-one connected—meaning the server must know how many clients may connect. This is unhandy for elastic applications because clients can dynamically connect to a server. Therefore, KRCORE VQP is many-to-one: after binding to an address, a VQP can dynamically accept new connections when receiving messages: `qpops_msgs` will return a list of (`src_qp`, `message`) pairs, where the `src_qp` is a VQP connected to the corresponding sender of the `message`.

Besides the extended API, KRCORE also supports common verbs data path API, e.g., `ibv_post_send`, `ibv_post_recv` and `ibv_poll_cq` (see §2.2). Figure 7 shows a simplified code example of sending a message from a client to a server with VQP. At the client, it can use `KRCORE_VQP` as a marker to create a VQP. After successfully connecting the VQP with `qconnect`, the client can call `ibv_post_send` to send the message.

Note that the VQP has the semantic as RCQP—meaning that they have reliability guarantees and support all RDMA operations (with various low-level optimizations).

4.2 Data structures

Hybrid QP pool. Each VQP (§4.1) is backed by a kernel-space virtual QP that has an identifier, a reference to a physical QP and virtualized counterparts of RDMA queues (see §2.2). The physical QP is selected from a hybrid QP pool with both DCQPs and RCQPs. The DCQPs are statically initialized upon boot time and RCQPs are created on-the-fly.

In principle, the pool only needs one DCQP to handle all the RDMA requests of the host. However, only using one DCQP introduces extra latency when sending concurrent requests to different servers. Specifically, if two requests targeting different hosts go over the same DCQP, the second must wait for an additional reconnection before RNIC can process it. This can be mitigated by increasing the DCQP pool size since reconstructions can run concurrently. Yet, the best choice of the pool size depends on the hardware setting (§5.2). On our platform, we choose 8 DCQPs in the pool.

To further prevent lock contention [26], we divide the pool on a per-CPU basis: Each VQP only virtualizes QPs from its local CPU's pool. This strategy is optimized for cases when each QP is exclusively used by one thread, a common pattern in RDMA applications [47, 55, 26, 17, 33]. In case

of thread migrations, KRCORE also re-virtualizes QPs in the background with a transparent QP transfer protocol (§4.6).

Meta Server. For steady and low-latency DCT metadata query, we replicate all the nodes’ metadata at a few global meta servers backed by DrTM-KV [58], a state-of-the-art RDMA-enabled KVS. Note that replicating all the DCT meta at one server is practical because they are extremely small (e.g., 17KB for a 1,000-server cluster).

The meta server stores a mapping between the RDMA address (key) and its corresponding DCT number and key (value). These key-value pairs can be queried via DrTM-KV with a few one-sided RDMA READs. Since sending one-sided requests also requires RDMA connections, each node pre-connects to nearby meta servers (e.g., one in the same rack) with RCQPs during boot time and thus, it can find the DCT metadata of a given server in several microseconds even under high load.

Optimization: DCCache. Observing that the DCT metadata is extremely small (12B), each node further caches them locally to save network round-trips querying the meta server. The metadata is suitable for caching because they are only invalidated when the corresponding host is down.

ValidMR and MRStore. To safely virtualize a physical QP to multiple VQPs, KRCORE additionally checks the validity of remote memory accesses to prevent QP state corruption (§4.4). These checks were originally done by the RNIC using the information stored in the NIC cache. Thus, we should also record them in KRCORE. We additionally bookkeep the registered memory regions (MR)s in ValidMR, which is also implemented with DrTM-KV. After the bookkeeping, KRCORE can query the local/remote ValidMRs to check the local/remote memory regions’ validity.

Like DCCache, we also cache the checked remote MR locally (in MRStore) to avoid extra round-trips. However, caching remote MRs may introduce consistency problems: unlike long-lived DCT metadata, MRs are managed by the applications and can be de-registered on-the-fly. To this end, KRCORE adopts a lease-based lightweight invalidation scheme: the cached MRs are periodically (e.g., 1 second) flushed. Upon de-registration, KRCORE waits for this period before freeing the MR.

4.3 Control path operations

KRCORE reuses initialized QPs upon VQP connection and creation, whose simplified pseudocode executed in the KRCORE kernel is shown in Algorithm 1.

`vqp_create` initializes the basic data structures of VQP—mainly allocating the software send and completion queues in the kernel. The physical QP assignment is delayed to the VQP connection (line 5) because we are unaware of the remote target during creation.

`vqp_connect` connects a VQP to a remote end by assigning a pre-initialized kernel-space QP (either RCQP or

Algorithm 1: VQP creation and connection

```

1: Function vqp_create (Q):
2:   Q.id ← allocate a free identifier
3:   Q.comp_queue ← allocate a software queue
4:   Q.recv_queue ← allocate a software queue
5:   Q.qp ← NULL ← Updated by qconnect
6: Function vqp_connect (Q, addr):
7:   if Q.qp == NULL then
8:     if addr in HybridQPPool.RC then
9:       Q.qp ← select in HybridQPPool.RC[addr]
10:    else
11:      Q.qp ← select in HybridQPPool.DC
12:      if addr not in DCCache then
13:        meta ← query nearby connected MetaServer
14:        add meta to DCCache
15:      Q.dct_meta ← meta

```

DCQP) to it. Given the remote *addr*, it first checks whether an RCQP is available in the HybridQPPool (line 8). If so, we choose an available QP and assign it to *Q.qp* (line 9). Otherwise, we select a DCQP (line 11). Note that all DCQPs in the pool are available because KRCORE can virtualize one physical QP to multiple VQPs (§4.4).

When assigning a DCQP to VQP, we need to fetch the remote end’s DCT metadata (line 12–15) if the metadata is not cached in the *DCCache*. We issue one-sided RDMA READs to the *MetaServer* to query it (line 13).

Background RCQP creations. To increase the likelihood of hitting an RCQP in the pool, KRCORE maintains background routines to sample frequently communicated nodes, create RCQPs for frequently communicated ones in the *HybridQPPool* and reclaim rarely used RCQPs. Currently, we choose a simple LRU strategy for the reclamation.

Other control path operations. Besides VQP creation and connection, other control path operations (e.g., memory registration, MR) have a straightforward implementation: we forward them to the corresponding verbs API and record the results in KRCORE. If necessary, we will also return the virtual handler of the recorded results to the user. Due to space limitations, we omit a detailed description.

4.4 Data path operations

As we have mentioned in §3.1, a key challenge in virtualizing a physical QP to multiple VQPs is preventing shared QP state corruption. Specifically, we must consider:

1. **Detecting malformed request.** An incorrect operation code or an invalid memory reference would transit a QP into error states. Since an error states QP cannot handle any RDMA requests, we must filter out malformed requests before posting them to the physical QP.
2. **Preventing NIC queue overflow.** The physical QP has a limited queue capacity. If the user overflows a QP, the QP will also enter an error state. Preventing queue overflow is challenging under sharing because it can overflow even if all the shared users correctly avoid the queue overflows.

Algorithm 2: kernel handler of `post_send` and `poll_cq`

```
1: Function post_send_virtualized(Q, wr_list):  
   < wr_list: the RDMA requests list  
   < Assumption: the size of wr_list is smaller  
   than Q.qp.sq.max_depth and Q.qp.cq.max_depth  
2: while Q.qp.sq.max_depth - Q.qp.uncomp_cnt <  
   wr_list.length do  
3:   | poll_inner(Q)  
4:   unsignaled_cnt ← 0  
5:   for req in wr_list do  
6:     | if req has invalid MR or invalid Op then  
7:       | return Error  
8:     | if req is signaled then  
9:       | Q.comp_queue.add(NotReady, req.wr_id)  
10:      | req.wr_id ← encode the pointer of Q and  
      | (unsignaled_cnt + 1)  
      | unsignaled_cnt ← 0  
11:     | else  
12:       | unsignaled_cnt += 1  
13:       | Q.qp.uncomp_cnt += 1  
14:   if last_req in wr_list is not signaled then  
15:     | mark last_req as signaled  
16:     | last_req.wr_id ← encode NULL and  
17:     | (unsignaled_cnt + 1)  
18:   return post_send(Q.qp, wr_list)  
19: Function poll_inner(Q):  
20:   wc ← poll_cq(Q.qp.cq)  
21:   if wc is ready then  
22:     | VQ, comp_cnt ← decode wc.wr_id  
23:     | Q.qp.uncomp_cnt -= comp_cnt  
24:     | if VQ is not NULL then  
25:       | VQ.comp_queue.head()[0] = Ready  
26: Function poll_cq_virtualized(Q):  
27:   poll_inner(Q)  
28:   if Q.comp_queue.has_head() and  
   Q.comp_queue.head()[0] is ready then  
29:     | user_wr_id ← Q.comp_queue.pop()[1]  
30:     | return READY, user_wr_id  
31:   return NULL, 0
```

The queue can be cleared via explicit signaling and polling. Nevertheless, we should poll as little as possible because they have overheads [24].

3. **Dispatching completion events.** The polled results of a physical QP can be from different VQPs. Therefore, we must correctly dispatch them to the targets, i.e., software queues of VQPs.

To this end, KRCORE will (1) check the request integrity before posting it to a shared QP; (2) inject necessary polls to the physical QP and (3) encode the VQP information in the request's `wr_id`—that will be returned upon request completion—to help the dispatch. Specifically, KRCORE executes `post_send_virtualized` and `poll_cq_virtualized` after the user calls `ibv_post_send` and `ibv_poll_cq`, respectively. Algorithm 2 shows their simplified pseudocode. For simplicity, we assume the request list (`wr_list`) depth is smaller than the QP capacity, which can be achieved by segmenting the request list before posting it.

post_send_virtualized. It first clears the physical QP's send and completion queues to prevent overflows

(line 2–3) via polling the physical completion queue (line 20). Polling is tricky when considering unsignaled requests—the requests that don't generate completion events. Their entries are freed until a later signaled request is polled. Thus, we must track how many requests a signaled one is responsible to clear (line 4 and line 13), and encode the number in `wr_id` (line 10). Therefore, after polling a completion we can determine the left spaces of queues (line 23). Further, if the last request is unsignaled, we signal it (line 15–17).

For each request, we also check whether it is malformed (line 6) and record the dispatch information for the signaled ones (line 9–10). Finally, we can safely post these requests to the physical QP (line 18).

For two-sided primitive, KRCORE must additionally notify the receiver the sender information. Otherwise, the receiver cannot create proper connections in `qpop_msgs`. Hence, we piggyback the sender's address in the message header (omitted in the algorithm).

poll_cq_virtualized. It first calls `poll_inner` to poll the physical QP events and dispatch the events to the proper VQPs according to the information recorded in the `wr_id` (lines 22–25). After the dispatch, it can check whether the virtualized QP has a completion event. KRCORE examines the head of the virtualized `comp_queue` and returns the head's `wr_id` to the application if the head exists.

Due to space reasons, we briefly describe other operations:

ibv_post_recv. This function registers the buffers to the VQP by recording them in the virtualized `recv_queue`.

qpop_msgs. It polls the physical QP's `recv_queue` and dispatches the received messages, similar to `poll_inner`. To hold in-coming messages, we pre-post message buffers to physical QP before virtualizing it to the applications. The challenge of pre-post is that the KRCORE doesn't know the exact payloads of the incoming messages. For now, we assume the pre-posted buffers can always hold the incoming message. §4.5 will describe how we cope with out-of-bound messages in detail. After receiving a message, we will check its destination VQP and copy it the user-registered buffer (from `ibv_post_recv`).

Besides receiving messages, `qpop_msgs` also creates a VQP connected to the sender (§4.1). The creation and connection follow the control path operations discussed in §4.3. To prevent the DCT metadata query, we further piggyback the metadata in the message header. Thus, `qpop_msgs` doesn't involve additional networking requests.

4.5 Zero-copy protocol for two-sided operations

The basic `qpop_msgs` (§4.4) has two issues. First, it incurs extra memory copies. Though the copy overhead is negligible for small messages (e.g., less than 1KB), it is non-trivial for the large ones (e.g., see results in Figure 9 (b)). Second, it cannot receive messages with payloads larger than the pre-posted buffers.

To this end, we adopt a zero-copy protocol to overcome the above issues. Intuitively, for large or out-of-bound messages, the receiver will use one-sided RDMA READ to read them to the user-registered buffers, inspired by existing RDMA-enabled RPC frameworks [48, 17]. Specifically, if the payload is larger than the kernel’s registered buffer, the sender will first send a small message containing the destination VQP ID, the source message address and its size. The receiver can then use one-sided RDMA READ to read the message directly to the user-registered buffer in a zero-copy way. The cost of sending an additional message is trivial for large messages because the network transfer will dominate the time.

4.6 Physical QP transfer protocol

KRCORE supports seamlessly changing the physical QP virtualized by a VQP to another. The challenge of doing so is how to preserve the RCQP’s FIFO property [7] of the VQP during transfer, i.e., after a request completes, all its previous requests are finished.

To ensure FIFO, upon the transfer starts, we first post a fake signaled RDMA request to the source QP and wait for its completion before the change. Meanwhile, we also notify the remote peers to transfer their physical QP. Otherwise, the VQP can no longer receive the remote end’s message. For correctness, we must wait for the remote acknowledgments before changing the physical QP at the sender.

5 Evaluation

We aim to answer the following questions during evaluations:

1. How fast is the KRCORE control plane (§5.1)?
2. What are the costs to the data plane (§5.2)?
3. How RDMA-aware applications that require elasticity can benefit from KRCORE (§5.3)?

Implementation. We implement KRCORE from scratch as a loadable Linux kernel (4.15) module, which has more than 10,000 LoC Rust code. It exports system calls via *ioctl* without modifying the kernel. To simplify user-kernel interactions, we further implement a 100 LoC C shim library atop *ioctl* to provide the interfaces described in §4.1. Finally, we port DCT to the kernel-space RDMA driver by adding around 250 LoC C code to the *mlx-ofed-4.9* driver: DCT is currently only implemented in the user-space RDMA drivers.

Testbed setup. We conduct experiments on a local rack-scale RDMA-capable cluster with ten nodes. Each node has two 12-core Intel Xeon E5-2650 v4 processors, 128GB DRAM and one ConnectX-4 MCX455A 100Gbps InfiniBand RNIC. All nodes are connected to a Mellanox SB7890 100Gbps InfiniBand Switch. Without explicit mention, we deploy one meta server for KRCORE.

Comparing targets. We compare KRCORE with user-space *verbs* (*verbs*) and LITE⁵. Original LITE has an unoptimized control plane: it uses a centralized cluster manager to

⁵<https://github.com/WukLab/LITE>

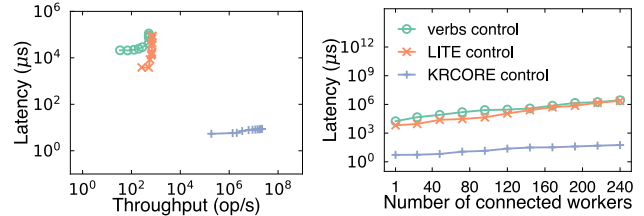


Fig. 8. The *qconnect* performance of KRCORE when using DCQP with DCT metadata uncached. (a) Connecting to a single server, and (b) establishing connections in a full-mesh fashion.

establish connections between servers and can only connect tens of RCQPs per second. Therefore, we further optimize it by enabling a decentralized QP connection scheme via RDMA’s unreliable datagram (UD). Our optimized version can achieve an optimal kernel-space RDMA control plane performance—it is now only bottlenecked by the hardware limits. §5.1 will describe this in more detail. Note that our optimization leaves the LITE data plane unchanged.

5.1 Control path performance

The evaluations for the control path focus on creating and connecting RDMA connections. The costs of the other operations in KRCORE (and *verbs*) are typically much smaller. For example, registering 4MB memory only takes 1.4μs in KRCORE. Therefore, we omit their results.

We use two synthetic workloads (single and full-mesh connection establishment) to evaluate the control path performance. The connection pool and DCCache of KRCORE are cleared before the evaluations. Otherwise, KRCORE only has system call overheads and is extremely small (0.9μs).

Single-connection establishment performance. We first evaluated the latency and throughput of establishing a single RDMA-capable connection to one server w.r.t. the number of clients. Figure 8 (a) reports the throughput-latency graph when increasing the number of clients from 1 to 240. From the figure we can see that KRCORE can have several orders of magnitude better performance than *verbs* and LITE. At one client, KRCORE can establish a connection in 5.4μs, while *verbs* and LITE take 15.7ms and 2ms, respectively. The performance gain of KRCORE comes from replacing the costly RDMA control path operations (analyzed in §2.3.1 and §2.3.2 in detail) with fast RDMA data path operations, i.e., two one-sided RDMA READs to the meta server. For LITE, it saves the driver loading cost but still needs to create and configure QP on its control path. At 240 clients, KRCORE can handle 22 million (M) connections per second, while *verbs* and LITE can only establish 712 RCQPs per second. They are both bottlenecked by the server creating hardware resources, while KRCORE always reuses existing ones to prevent these overheads.

Full-mesh connection establishment performance. Besides establishing a single connection, creating full-mesh connections at a set of workers is common in elastic applica-

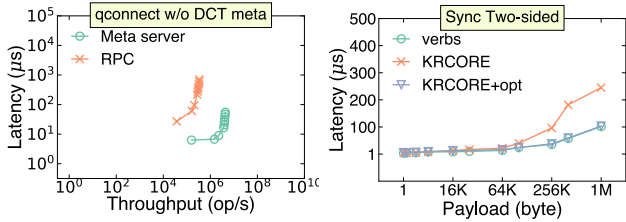


Fig. 9. (a) Performance comparisons of different DCT meta query methods, and (b) the effects of zero-copy protocol (KRCORE+opt) of KRCORE two-sided operations.

tions, e.g., burst-parallel serverless workloads [51]. Specifically, each worker should connect to the others and vice versa. Figure 8 (b) presents the full-mesh performance by varying the number of involved workers. In general, KRCORE can reduce 99% of the full-mesh creation time regardless of the worker number, thanks to the orders of magnitude faster single-connection establishment performance (see Figure 8 (a)). For example, KRCORE connected 240 workers in 81 μ s, while verbs and LITE used 2.7 secs and 2.3 secs, respectively. These results suggest that KRCORE can handle complex control path operations well.

Benefit of the meta server. A key design choice of KRCORE is to use an RDMA-based meta server to store DCT meta. Figure 9 (a) illustrates the benefit of this design using the single-connection establishment workload of Figure 8 (a). The baseline (RPC) uses a kernel-space FaSST [26] RPC for the querying. FaSST is the state-of-the-art RDMA-based RPC that builds on RDMA’s unreliable datagram. It also has no control plane overhead in the kernel because UD is connectionless. To save CPU resources, we only deploy one kernel thread to handle the queries. We can see that a meta server design achieves an 11.8X better throughput and up to 13X query latency compared with RPC. The RPC design is bottlenecked by the server CPU for handling DCT queries, while the RDMA-based meta server bypasses the CPU with one-sided RDMA.

5.2 Data path performance

KRCORE trades data path performance for a faster control plane. We first use a set of microbenchmarks to evaluate these overheads using two communication patterns: **sync** and **async**. In the sync mode, each client issues RDMA requests to one server in a run-to-completion way, aiming to achieve low latency [17, 47]. For async, each client posts requests in batches to achieve the peak throughput [57, 24, 25]. Without explicit mention, the workloads are inbound, i.e., multiple clients sending RDMA requests to one server. We reported the aggregated throughput of clients and their average latency.

One-sided operations. Figure 10 presents the one-sided data path performance of KRCORE when it virtualizes from DCQP (KRCORE(DC)) and RCQP (KRCORE(RC)), and compare them to verbs⁶. During the experiment, each client

⁶LITE’s data path API is different so we compare to it separately.

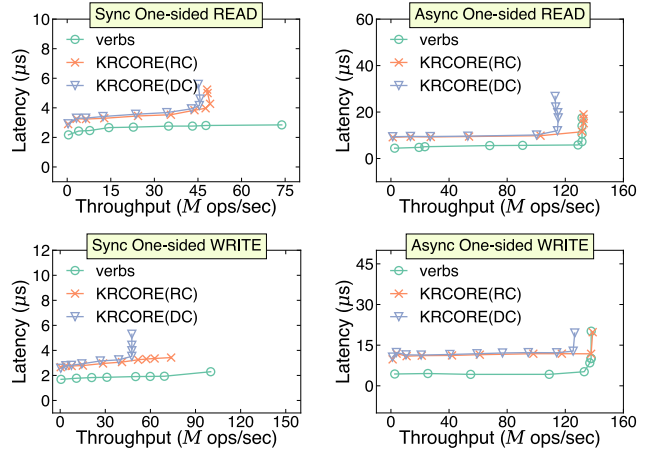


Fig. 10. The one-sided RDMA performance.

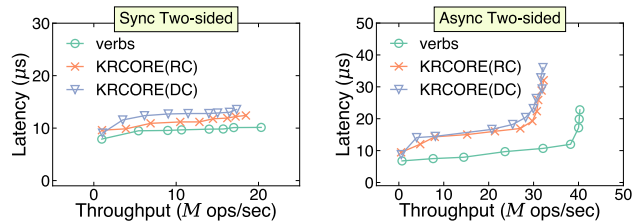


Fig. 11. The two-sided RDMA performance of KRCORE.

issued 8B random requests to the server, and we varied the number of clients from 1 to 240.

(1) Sync. For one-sided RDMA READ in Figure 10 (a), the latency of KRCORE (DC) and (RC) is 27%–46% and is 25%–41% higher than verbs. The additional latency of KRCORE under sync mode is dominated by the system call cost. On our hardware, we measure a $\sim 1\mu$ s overhead communicating with the kernel. For reference, when using one client, the latency of KRCORE (RC) is 3.15μ s, and the verbs is 2.15μ s. Another observation is that adopting DCQP has little latency overhead in the sync mode as DC reconnection is extremely fast. For example, the latency of KRCORE (DC) under one client is 3.24μ s. The results of one-sided RDMA WRITE in Figure 10 (c) are similar to the READ.

(2) Async. For one-sided RDMA READ in Figure 10 (b), KRCORE (RC) can achieve a similar peak throughput as verbs (138M reqs/sec) when using 240 clients. With the same configuration, KRCORE (DC) is 14% slower (118 M reqs/sec). KRCORE (RC) and verbs are both bottlenecked by the server RNIC, while KRCORE (DC) is slower due to extra DCT processing at the RNIC. For one-sided RDMA WRITE in Figure 10 (d), the results are similar: KRCORE (RC) and verbs achieve a peak throughput of 145M reqs/sec while KRCORE (DC) is 8.9% lower (132M reqs/sec).

Two-sided operations. Figure 11 presents the two-sided throughput and latency of KRCORE w.r.t. to the number of clients (1 to 240). Each client sends an 8B request to the server in an echo fashion: after receiving a request, the server

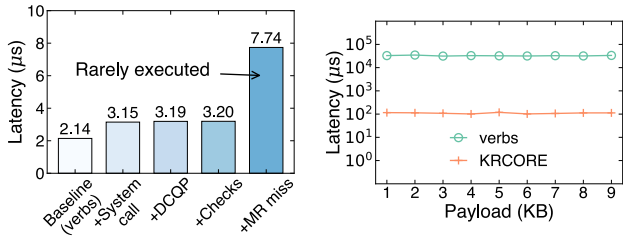


Fig. 12. (a) A factor analysis of the data path cost introduced by KRCORE using one-sided RDMA READ. (b) The performance of KRCORE in data transfer benchmark of serverless computing.

will send the request back, and the client will issue another request after getting the acknowledgment. The server utilizes all cores (24 threads) to handle these requests.

(1) Sync. In this mode, the performance comparisons are similar to one-sided RDMA: compared with verbs, KRCORE (RC) and (DC) have 4–21% and 14–31% higher latency, respectively. The KRCORE overheads added to two-sided RDMA are also dominated by the user-kernel interactions. For example, at one client, one KRCORE (RC) echo takes $9.6\mu\text{s}$ while verbs takes $7.9\mu\text{s}$. Compared to one-sided RDMA, the absolute latency gap is larger. KRCORE two-sided has an additional system call overhead: the server needs to enter the kernel to receive a message.

(2) Async. Unlike one-sided RDMA, KRCORE cannot achieve the same peak inbound throughput (when using 240 clients) as verbs for two-sided RDMA: it is 20% slower than verbs: which can only achieve 33.7M reqs/sec regardless of RC or DC. In comparison, verbs can achieve 42.3M reqs/sec. The extra bottleneck comes from CPU processing costs at the server due to user-kernel interactions. As a result, KRCORE cannot saturate the RNIC’s high performance. This also explains why KRCORE has a similar performance when using RC and DC.

Effects of zero-copy optimization. We next examine the costs of memory copy—that KRCORE uses to dispatch messages between virtual QPs—to the two-sided operations. We further demonstrate how we mitigate it with a zero-copy protocol (§4.5). Figure 9 (b) shows the two-sided echo latency when using one client to communicate with the server w.r.t. the payload size. We can see that the memory copy cost is negligible for small transfers ($\leq 16\text{KB}$) but is significant for large messages. Specifically, when transferring $> 16\text{KB}$ messages, the latency of KRCORE is 1.45–3.1X higher than verbs. To this end, the zero-copy optimization (KRCORE+opt) reduces the overheads to 0.08–0.23X when transferring $\geq 16\text{KB}$ messages.

Factor analysis. Figure 12(a) conducts a factor analysis to show the detailed data path costs of KRCORE in a sync one-sided RDMA READ request. The main observations are: (1) The biggest cost to data path operations is additional RDMA requests to check the MR validity when the remote MR information is not cached locally (+MR miss, takes

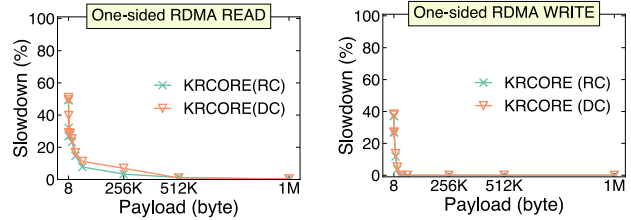


Fig. 13. The slowdown of KRCORE compared to verbs on one-sided RDMA READ (a) and WRITE (b), respectively.

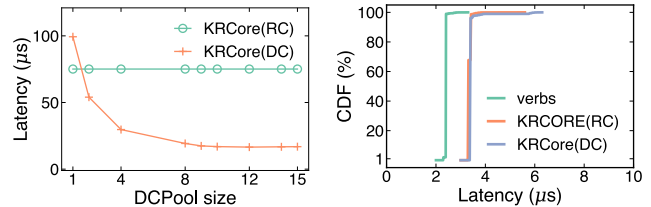


Fig. 14. (a) The impacts of DCQP pool size. (b) The CDF of latency of sending RDMA requests to different servers.

$4.5\mu\text{s}$). Note the checks are rare because KRCORE always caches the checked MR after a miss.

(2) For normal requests without MR checks, system call dominates the overheads (+System call), resulting in $1\mu\text{s}$ latency increase ($3.15\mu\text{s}$ vs. $2.14\mu\text{s}$). Other costs—including using DCQP (+DCQP) and KRCORE check to prevent QP state corruptions (+Checks, see §4.4) are trivial (less than $0.5\mu\text{s}$).

Impacts of payload size to one-sided RDMA. The overhead of KRCORE becomes smaller for one-sided RDMA with a larger payload, since transferring data through the network dominates the time. Figure 13 reports the slowdown compared to verbs on different request payloads. We measure the latency of sync one-sided RDMA with one client. For one-sided RDMA READ, the overhead is negligible for larger than 256KB reads ($< 7\%$). For WRITE, the overhead is negligible for larger than 8KB payloads.

Impacts of DCQP pool size. A larger DCQP pool is typically better for concurrently sending requests to different machines (§4.2). Figure 14 (a) reports the latency when sending a batch of 64 one-sided RDMA READs to different targets at one client with different pool sizes. The targets are randomly selected in 10 machines. We can see that when the pool only has one DCQP, KRCORE (DC) has a 1.32X higher latency (99 vs. $75\mu\text{s}$) than KRCORE (RC), since requests to the same QP are processed sequentially with reconnections. Increasing the pool size can significantly improve the latency. Interestingly, when the pool size is larger than 2, DC outperforms RC by 28–78%. RC needs 64 different connections to send these requests, and it has to do 63 additional polls than DC.

Tail latency. Figure 14 (b) reports the tail latency when using 50 clients sending sync one-sided RDMA READ to 5 servers. Under such a fan-out scenario, KRCORE (DC) has a higher tail latency than the others due to extra round-trips caused by DC reconnections. The 99.9% latency of verbs,

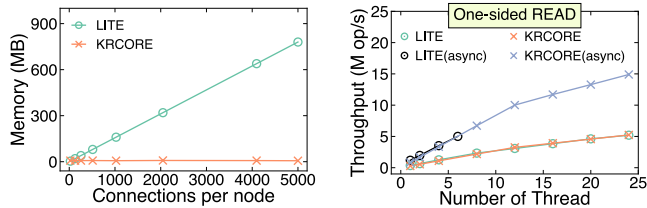


Fig. 15. (a) A comparison of memory usage on connections: KRCORE caches all DCT metadata, while LITE caches all RCQPs. (b) A comparison of data path performance when KRCORE uses DCQP.

KRCORE (RC) and KRCORE (DC) are $2.8\mu\text{s}$, $3.8\mu\text{s}$ and $6\mu\text{s}$, respectively.

Comparison to LITE. Finally, we show that KRCORE can achieve a similar (or better) data path performance than LITE with smaller memory usage.

(1) Memory. Figure 15 (a) shows the memory used for caching RDMA connections. In general, KRCORE consumes orders of magnitude smaller memory when supporting the same number of connections. For example, to maintain 5,000 connections, LITE consumes 780MB of memory, even without counting the memory of message queues (1.5GB if counted). In comparison, KRCORE only consumes 6.3MB of memory because it just maintains a (small) constant number of DCQP (48), and each DCT metadata only consumes 12B.

(2) Performance. Figure 15 (b) further compares the throughput when issuing 64B random one-sided RDMA READ from one node to others. We configure both systems to deploy a pool of 32 connections, preventing LITE from encountering RCQP scalability issues [26]. KRCORE uses DCQP for its connections. For sync, we can see that KRCORE is up to 20% slower than LITE due to performance issues of DCQP. On the other hand, KRCORE achieves a 3X higher peak async throughput (15.6M/sec vs. 5.2M/sec) in the async mode. LITE has a limited peak performance because it fails to run with more than 6 threads. LITE doesn't prevent QP queue overflows (see issue #3 in §2.3.2), so it will trigger QP errors for more than 6 threads. KRCORE handles overflows well (§4.4) and can thus, scale to more threads.

5.3 Application performance

5.3.1 Scaling RACE Hashing

Overview and setup. RACE hashing [67] is a production RDMA-enabled disaggregated key-value store. We chose it as our case study because it requires elastically—a demand not commonly found in existing RDMA-based key-value stores. At a high level, RACE separates the storage nodes and computing nodes by RDMA, where the computing nodes execute key-value store requests by issuing one-sided RDMA requests to the storage nodes. RACE further allocates computing nodes on-demand to cope with various workloads in a resource-efficient way, where the newly started nodes need dynamically establish RDMA connections to memory nodes. To improve performance, it embraces a set of low-level RDMA-

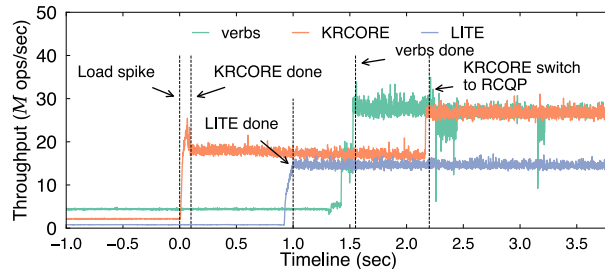


Fig. 16. Under load spikes, KRCORE can quickly bootstrap computing nodes for RACE Hashing [67].

aware optimizations—e.g., doorbell batching [25] that are tailed to RDMA's low-level verbs interface.

Since RACE is not open-sourced, we implement a simplified version atop of verbs, LITE and KRCORE, respectively. We have calibrated that the performance is close to their reported ones. For example, RACE reports a peak 24M req/sec Get throughput on ConnectX-5 under YCSB-C [11]. Our (verbs) version can achieve 27M req/sec with more machines (8 vs. 5) on a similar RNIC (ConnectX-4).

Performance under load spikes. Our evaluating workload contains a load spike commonly found in real-world applications [8, 28, 2]. Under spikes, RACE allocates more computing processes to increase performance. During process startups, KRCORE can reduce its bootstrap time thanks to its fast control plane.

Figure 16 shows the timelines of RACE atop of verbs, LITE and KRCORE under load spikes, respectively. The spikes happen at time 0, and RACE forks 180 new processors to handle it. When using KRCORE, RACE can finish the startup in 244ms, 83% and 76% faster than verbs (1.4 seconds) and LITE (1 second), respectively. KRCORE is bottlenecked by OS creating worker processors. On the other hand, LITE and verbs are bottlenecked by RDMA's slow control path (§2.3). A fast boot further reduces the tail latency: during time 0-3, KRCORE has a 4.9X lower 99% latency than verbs.

Benefit of virtualizing a low-level RDMA API. KRCORE virtualizes a low-level RDMA (e.g., `ibv_post_send`), and thus, it can transparently apply existing RDMA-aware optimizations (see Issue #3 in §2.3.2). This leads to better performance of KRCORE on RACE compared to LITE: as shown in Figure 16, KRCORE has a 1.73X higher peak throughput (26M reqs/sec vs. 15M reqs/sec) than LITE after time 3.

Benefit of virtualizing hybrid QPs. As shown in Figure 16, using RCQP (e.g., after time 3) brought 1.4X (26M vs. 18M req/sec) throughput improvements to KRCORE, achieving a similar performance as verbs (26M reqs/sec). This is because RACE issues RDMA requests asynchronously, and KRCORE's RC async peak throughput is similar to verbs (see Figure 10 (b)). Further, we can see the overhead of switching from DCQP to RCQP is negligible (at time 2.2). However, there is a lag for detecting the switch because KRCORE needs time to collect the necessary information to decide which RC-

QPs to create.

5.3.2 Accelerating data transfer in serverless computing

Finally, we show that KRCORE can improve the communication performance between functions in serverless computing. We use an RDMA-version of data transfer testcase in ServerlessBench [62] (TestCase5), a state-of-the-art Serverless benchmark suite. This testcase measures the data transfer time between two serverless functions. The experiment runs on Fn [43], a popular open-source serverless platform.

Figure 12 (b) reports the time to pass a message w.r.t. the payload size when using verbs and KRCORE, respectively. The receiver function runs in a separate machine using a Docker container after the sender finishes execution. We use warm start to techniques [40] to reduce the control plane costs of starting containers. From the figure we can see KRCORE reduces the data transfer latency of verbs by 99% when transferring 1KB to 9KB bytes. The performance improvements are mainly due to the reduced RDMA control path of KRCORE, which we have extensively analyzed in §5.1.

6 Discussion

Trade-offs of a kernel-space solution. KRCORE chooses kernel-space RDMA for a microsecond-scale control plane (5,900X faster than verbs). Though it retains most benefits of RDMA (e.g., zero-copy), we sacrifice kernel-bypassing benefit and thus, result in a slower data path (up to 75% slowdown). We argue that such cost is acceptable to many elastic applications. First, the application usually issues a few networking requests. For example, the functions in ServerlessBench [62] and SeBS [12] only issue one request to read/write remote data on average. Second, the control path overhead (ms-scale) is commonly orders of magnitude higher than the cumulative data path overhead (μ s-scale), see Figure 3. Finally, existing work (i.e., LITE [53]) also showed that kernel-space RDMA is efficient for many datacenter applications.

Other RNICs. Our analysis focuses on Mellanox ConnectX-4 Infiniband RNIC. Nevertheless, we argue the cost is unlikely to reduce due to hardware upgrades or different RDMA implementations (e.g., RoCE) since the cost is dominated by configuring the NIC resources. For example, we also evaluate the control path performance on ConnectX-6, where the user-space driver still takes 17ms for creating and connecting QP, similar to the ConnectX-4 we evaluated (15.7ms, see Figure 3).

KRCORE in virtualized environments. We currently focus on accelerating RDMA control plane with host networking mode. Using RDMA in virtual machines or virtualized RDMA network [30, 20] is also popular in the cloud. We believe the principles and methodologies of KRCORE are also applicable in these environments. For example, Freeflow [30] is an RDMA virtualization framework designed for containerized clouds. It leverages par-virtualization that intercepts virtualized RDMA requests to a software router. We can inte-

grate our hybrid connection pool to the router to support a fast control plane atop of it. We plan to investigate applying KRCORE in virtualized environments in the future.

7 Related Work

RDMA libraries. Many user-space RDMA libraries exist [32, 4, 3, 36, 64], e.g., MPI, UCX [4], rsocket [3]. They can hardly provide a fast control plane because they are all based on verbs. LITE [53] is the only kernel-space RDMA library and is the closest to our work. We have extensively analyzed the issues when deploying LITE in elastic computing (§2.3.2) and how KRCORE addresses them (§3—§4).

DCT-aware and hybrid-transport systems. Several works used DCT to improve the performance and scalability of RDMA-enabled systems [49, 41]. Subramoni et al. [49] showed that DCT could provide comparable performance to RC while reducing memory consumption for MPI applications. Meanwhile, several works leveraged a hybrid-transport design to overcome the shortcoming of a single transport [31, 23]. For instance, Jose et al. [23] utilized UD to reduce the memory consumption of RC in Memcached.

RDMA-enabled applications. KRCORE continues the line of research on accelerating systems with RDMA, from key-value stores [38, 55, 67, 24, 13, 39], far-memory data structures [45, 6, 44], RPC frameworks [48, 26, 9, 27], replication systems [5, 42, 54, 29], distributed transactions [58, 46, 14, 10, 57, 65, 56], graphs [47, 59, 61, 18] and distributed file systems [66, 33, 60], just to name a few. Most of these systems do not target elastic computing, but we believe there are opportunities for applying them in such a setting. In such scenarios, they can benefit from KRCORE.

8 Conclusion

This paper presents KRCORE, a μ s-scale RDMA control plane for RDMA-enabled applications that require elasticity. By retrofitting RDMA dynamic connected transport with kernel-space QP virtualization, we show that it is possible to eliminate most RDMA control path costs on commodity RNICs. Meanwhile, the data path costs introduced by KRCORE are acceptable for many elastic applications. Our experimental results confirm the efficacy of KRCORE.

9 Acknowledgment

We sincerely thank the anonymous shepherd and reviewers for their insightful suggestions. We also thank Dingji Li for discussing how to apply KRCORE to virtual machines, Xiating Xie for improving the figures and Sijie Shen, Zhiyuan Dong, Rongxin Chen and Yuhan Yang for their valuable feedback. This work was supported in part by the National Key Research & Development Program of China (No. 2020YFB2104100), the National Natural Science Foundation of China (No. 61732010, 61925206) and Shanghai AI Laboratory.

References

- [1] Dynamically connected transport. https://www.openfabrics.org/images/eventpresos/workshops2014/DevWorkshop/presos/Monday/pdf/05_DC_Verbs.pdf, 2014.
- [2] Daily Deals and Flash Sales: All the Stats You Need to Know. <http://socialmarketingfella.com/daily-deals-flash-sales-stats-need-know/>, 2016.
- [3] rsocket(7) - Linux man page. <https://linux.die.net/man/7/rsocket>, 2021.
- [4] Unified communication x. <https://openucx.org>, 2021.
- [5] AGUILERA, M. K., BEN-DAVID, N., GUERRAOU, R., MARATHE, V. J., XYGKIS, A., AND ZABLOTCHI, I. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 599–616.
- [6] AGUILERA, M. K., KEETON, K., NOVAKOVIC, S., AND SINGHAL, S. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019* (2019), ACM, pp. 120–126.
- [7] ASSOCIATION., I. T. Infiniband architecture specification. <https://cw.infinibandta.org/document/dl/7859>, 2015.
- [8] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012* (2012), P. G. Harrison, M. F. Arlitt, and G. Casale, Eds., ACM, pp. 53–64.
- [9] CHEN, Y., LU, Y., AND SHU, J. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019* (2019), G. Candea, R. van Renesse, and C. Fetzer, Eds., ACM, pp. 19:1–19:14.
- [10] CHEN, Y., WEI, X., SHI, J., CHEN, R., AND CHEN, H. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016* (2016), C. Cadar, P. R. Pietzuch, K. Keeton, and R. Rodrigues, Eds., ACM, pp. 26:1–26:17.
- [11] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010* (2010), J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, Eds., ACM, pp. 143–154.
- [12] COPIK, M., KWASNIEWSKI, G., BESTA, M., PODSTAWSKI, M., AND HOEFLER, T. Sebs: a serverless benchmark suite for function-as-a-service computing. In *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021* (2021), K. Zhang, A. Gherbi, N. Venkatasubramanian, and L. Veiga, Eds., ACM, pp. 64–78.
- [13] DRAGOJEVIC, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (2014), R. Mahajan and I. Stoica, Eds., USENIX Association, pp. 401–414.
- [14] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (New York, NY, USA, 2015), SOSP'15, ACM*, pp. 54–70.
- [15] DU, D., YU, T., XIA, Y., ZANG, B., YAN, G., QIN, C., WU, Q., AND CHEN, H. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020* (2020), J. R. Larus, L. Ceze, and K. Strauss, Eds., ACM, pp. 467–481.
- [16] FACEBOOK. Introducing Bryce Canyon: Our next-generation storage platform. <https://engineering.fb.com/2017/03/08/data-center-engineering/introducing-bryce-canyon-our-next-generation-storage-platform/>, 2017.
- [17] GAO, Y., LI, Q., TANG, L., XI, Y., ZHANG, P., PENG, W., LI, B., WU, Y., LIU, S., YAN, L., FENG, F., ZHUANG, Y., LIU, F., LIU, P., LIU, X., WU, Z., WU, J., CAO, Z., TIAN, C., WU, J., ZHU, J., WANG, H., CAI, D., AND WU, J. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021* (2021), J. Mickens and R. Teixeira, Eds., USENIX Association, pp. 519–533.
- [18] GUO, C., CHEN, H., ZHANG, F., AND LI, C. Distributed join algorithms on multi-cpu clusters with gpudirect RDMA. In *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05-08, 2019* (2019), ACM, pp. 65:1–65:10.
- [19] GUO, C., WU, H., DENG, Z., SONI, G., YE, J., PADHYE, J., AND LIPSHTeyn, M. RDMA over commodity ethernet at scale. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016* (2016), M. P. Barcellos, J. Crowcroft, A. Vahdat, and S. Katti, Eds., ACM, pp. 202–215.
- [20] HE, Z., WANG, D., FU, B., TAN, K., HUA, B., ZHANG, Z., AND ZHENG, K. Masq: RDMA for virtual private cloud. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020* (2020), H. Schulzrinne and V. Misra, Eds., ACM, pp. 1–14.
- [21] JIA, Z., AND WITCHEL, E. Boki: Stateful serverless computing with shared logs. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021* (2021), R. van Renesse and N. Zeldovich, Eds., ACM, pp. 691–707.

- [22] JONAS, E., SCHLEIER-SMITH, J., SREEKANTI, V., TSAI, C., KHANDELWAL, A., PU, Q., SHANKAR, V., CARREIRA, J., KRAUTH, K., YADWADKAR, N. J., GONZALEZ, J. E., POPA, R. A., STOICA, I., AND PATTERSON, D. A. Cloud programming simplified: A berkeley view on serverless computing. *CoRR abs/1902.03383* (2019).
- [23] JOSE, J., SUBRAMONI, H., KANDALLA, K. C., WASI-UR-RAHMAN, M., WANG, H., NARRAVULA, S., AND PANDA, D. K. Scalable memcached design for infiniband clusters using hybrid transports. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012, Ottawa, Canada, May 13-16, 2012* (2012), IEEE Computer Society, pp. 236–243.
- [24] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014* (2014), F. E. Bustamante, Y. C. Hu, A. Krishnamurthy, and S. Ratnasamy, Eds., ACM, pp. 295–306.
- [25] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016* (2016), A. Gulati and H. Weatherspoon, Eds., USENIX Association, pp. 437–450.
- [26] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 185–201.
- [27] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019* (2019), J. R. Lorch and M. Yu, Eds., USENIX Association, pp. 1–16.
- [28] KHANDELWAL, A., AGARWAL, R., AND STOICA, I. Blowfish: Dynamic storage-performance tradeoff in data stores. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016* (2016), K. J. Argyraki and R. Isaacs, Eds., USENIX Association, pp. 485–500.
- [29] KIM, D., MEMARIPOUR, A. S., BADAM, A., ZHU, Y., LIU, H. H., PADHYE, J., RAINDEL, S., SWANSON, S., SEKAR, V., AND SESHAN, S. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018* (2018), S. Gorinsky and J. Topolcai, Eds., ACM, pp. 297–312.
- [30] KIM, D., YU, T., LIU, H. H., ZHU, Y., PADHYE, J., RAINDEL, S., GUO, C., SEKAR, V., AND SESHAN, S. Freeflow: Software-based virtual RDMA networking for containerized clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019* (2019), J. R. Lorch and M. Yu, Eds., USENIX Association, pp. 113–126.
- [31] KOOP, M. J., JONES, T., AND PANDA, D. K. Mvapi-chaptus: Scalable high-performance multi-transport MPI over infiniband. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008* (2008), IEEE, pp. 1–12.
- [32] LI, B., CUI, T., WANG, Z., BAI, W., AND ZHANG, L. Socks-direct: datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019* (2019), J. Wu and W. Hall, Eds., ACM, pp. 90–103.
- [33] LU, Y., SHU, J., CHEN, Y., AND LI, T. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017* (2017), D. D. Silva and B. Ford, Eds., USENIX Association, pp. 773–785.
- [34] MA, T., MA, T., SONG, Z., LI, J., CHANG, H., CHEN, K., JIANG, H., AND WU, Y. X-RDMA: effective RDMA middleware in large-scale production environments. In *2019 IEEE International Conference on Cluster Computing, CLUSTER 2019, Albuquerque, NM, USA, September 23-26, 2019* (2019), IEEE, pp. 1–12.
- [35] MELLANOX. <https://www.mellanox.com/sites/default/files/doc-2020/pb-connectx-6-enic.pdf>, 2021.
- [36] MELLANOX. <https://github.com/Mellanox/libvma>, 2021.
- [37] MELLANOX. Connect-IB product brief. https://www.mellanox.com/related-docs/prod_adapter_cards/PB_Connect-IB.pdf, 2021.
- [38] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013* (2013), A. Birrell and E. G. Sirer, Eds., USENIX Association, pp. 103–114.
- [39] MITCHELL, C., MONTGOMERY, K., NELSON, L., SEN, S., AND LI, J. Balancing CPU and network in the cell distributed b-tree store. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016* (2016), A. Gulati and H. Weatherspoon, Eds., USENIX Association, pp. 451–464.
- [40] OAKES, E., YANG, L., ZHOU, D., HOUCK, K., HARTE, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. SOCK: rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018* (2018), H. S. Gunawi and B. Reed, Eds., USENIX Association, pp. 57–70.
- [41] PARK, J., AND YEOM, H. Y. Design and implementation of software-based dynamically connected transport. In *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W), Trento, Italy, September 3-7, 2018* (2018), IEEE, pp. 58–64.
- [42] POKE, M., AND HOEFLER, T. DARE: high-performance state machine replication on RDMA networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2015, Portland, OR, USA, June 15-19, 2015* (2015), T. Kielmann, D. Hildebrand, and M. Tauber, Eds., ACM, pp. 107–118.

- [43] PROJECT, F. <https://fnproject.io>, 2021.
- [44] REDA, W., CANINI, M., KOSTIC, D., AND PETER, S. RDMA is turing complete, we just did not know it yet! *CoRR abs/2103.13351* (2021).
- [45] RUAN, Z., SCHWARZKOPF, M., AGUILERA, M. K., AND BELAY, A. AIFM: high-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 315–332.
- [46] SHAMIS, A., RENZELMANN, M., NOVAKOVIC, S., CHATZOPOULOS, G., DRAGOJEVIC, A., NARAYANAN, D., AND CASTRO, M. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019* (2019), P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds., ACM, pp. 433–448.
- [47] SHI, J., YAO, Y., CHEN, R., CHEN, H., AND LI, F. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 317–332.
- [48] SU, M., ZHANG, M., CHEN, K., GUO, Z., AND WU, Y. RFP: when RPC is faster than server-bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017* (2017), G. Alonso, R. Bianchini, and M. Vukolic, Eds., ACM, pp. 1–15.
- [49] SUBRAMONI, H., HAMIDOUCHE, K., VENKATESH, A., CHAKRABORTY, S., AND PANDA, D. K. Designing MPI library with dynamic connected transport (DCT) of infiniband: Early experiences. In *Supercomputing - 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings* (2014), J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds., vol. 8488 of *Lecture Notes in Computer Science*, Springer, pp. 278–295.
- [50] THE TRANSACTION PROCESSING COUNCIL. TPC-C Benchmark V5.11. <http://www.tpc.org/tpcc/>.
- [51] THOMAS, S., AO, L., VOELKER, G. M., AND PORTER, G. Particle: ephemeral endpoints for serverless networking. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020* (2020), R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds., ACM, pp. 16–29.
- [52] TSAI, S., SHAN, Y., AND ZHANG, Y. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020* (2020), A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 33–48.
- [53] TSAI, S.-Y., AND ZHANG, Y. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 306–324.
- [54] WANG, C., JIANG, J., CHEN, X., YI, N., AND CUI, H. APUS: fast and scalable paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017* (2017), ACM, pp. 94–107.
- [55] WEI, X., CHEN, R., AND CHEN, H. Fast rdma-based ordered key-value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 117–135.
- [56] WEI, X., CHEN, R., CHEN, H., WANG, Z., GONG, Z., AND ZANG, B. Unifying timestamp with transaction ordering for MVCC with decentralized scalar timestamp. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021* (2021), J. Mickens and R. Teixeira, Eds., USENIX Association, pp. 357–372.
- [57] WEI, X., DONG, Z., CHEN, R., AND CHEN, H. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018* (2018), A. C. Arpaci-Dusseau and G. Voelker, Eds., USENIX Association, pp. 233–251.
- [58] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015* (2015), E. L. Miller and S. Hand, Eds., ACM, pp. 87–104.
- [59] XIE, X., WEI, X., CHEN, R., AND CHEN, H. Pragh: Locality-preserving graph traversal with split live migration. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019* (2019), D. Malkhi and D. Tsafir, Eds., USENIX Association, pp. 723–738.
- [60] YANG, J., IZRAELEVITZ, J., AND SWANSON, S. Orion: A distributed file system for non-volatile main memory and rdma-capable networks. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019* (2019), A. Merchant and H. Weatherspoon, Eds., USENIX Association, pp. 221–234.
- [61] YAO, Z., CHEN, R., ZANG, B., AND CHEN, H. Wukong+: Fast and concurrent RDF query processing using rdma-assisted GPU graph exploration. *IEEE Trans. Parallel Distributed Syst.* 33, 7 (2022), 1619–1635.
- [62] YU, T., LIU, Q., DU, D., XIA, Y., ZANG, B., LU, Z., YANG, P., QIN, C., AND CHEN, H. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (New York, NY, USA, 2020), SoCC '20, Association for Computing Machinery, p. 30–44.
- [63] ZHANG, H., CARDOZA, A., CHEN, P. B., ANGEL, S., AND LIU, V. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 1187–1204.
- [64] ZHANG, I., RAYBUCK, A., PATEL, P., OLYNYK, K., NELSON, J., LEIJA, O. S. N., MARTINEZ, A., LIU, J., SIMPSON, A. K., JAYAKAR, S., PENNA, P. H., DEMOULIN, M., CHOUDHURY, P., AND BADAM, A. The demikernel datapath OS architecture for microsecond-scale datacenter systems. In

SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021 (2021), R. van Renesse and N. Zeldovich, Eds., ACM, pp. 195–211.

- [65] ZHANG, M., HUA, Y., ZUO, P., AND LIU, L. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association, pp. 51–68.
- [66] ZHU, B., CHEN, Y., WANG, Q., LU, Y., AND SHU, J. Octopus⁺: An rdma-enabled distributed persistent memory file system. *ACM Trans. Storage* 17, 3 (2021), 19:1–19:25.
- [67] ZUO, P., SUN, J., YANG, L., ZHANG, S., AND HUA, Y. One-sided rdma-conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (July 2021), USENIX Association, pp. 15–29.

A Artifact Appendix

Abstract. The artifact provides the source code and scripts to reproduce the experimental results from the USENIX ATC 2022 paper—"KRCORE: A Microsecond-scale RDMA Control Plane for Elastic Computing". KRCORE is a kernel-space RDMA solution that provides fast RDMA connection setups to user-space applications.

Scope. The artifact can be used to reproduce the evaluations in §5. It can also benefit the development of kernel-space RDMA-enabled applications.

Contents. The artifact contains the source code, the instructions for building and installation, and instructions for running the experiments in §5. All the above instructions can be found according to the steps in the `README.md` at the root directory of the artifact.

Hosting. The artifact is hosted on <https://github.com/SJTU-IPADS/krcore-artifacts> under the `main` branch with commit version `7ba3bf6`.



Zero-Change Object Transmission for Distributed Big Data Analytics

Mingyu Wu^{1,2}, Shuaiwei Wang¹, Haibo Chen^{1,3}, and Binyu Zang^{1,3}

¹Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University

²Shanghai AI Laboratory

³Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

Abstract

Distributed big-data analytics heavily rely on high-level languages like Java and Scala for their reliability and versatility. However, those high-level languages also create obstacles for data exchange. To transfer data across managed runtimes like Java Virtual Machines (JVMs), objects should be transformed into byte arrays by the sender (*serialization*) and transformed back into objects by the receiver (*deserialization*). The object serialization and deserialization (OSD) phase introduces considerable performance overhead. Prior efforts mainly focus on optimizing some phases in OSD, so object transformation is still inevitable. Furthermore, they require extra programming efforts to integrate with existing applications, and their transformation also leads to duplicated object transmission. This work proposes Zero-Change Object Transmission (ZCOT), where objects are directly copied among JVMs without any transformations. ZCOT can be used in existing applications with minimal effort, and its object-based transmission can be used for deduplication. The evaluation on state-of-the-art data analytics frameworks indicates that ZCOT can greatly boost the performance of data exchange and thus improve the application performance by up to 23.6%.

1 Introduction

High-level languages like Java and Scala are welcomed in areas like big-data analytics thanks to their reliable and versatile managed runtime environment. However, the abstraction provided by the managed runtime also introduces performance overhead, especially for data exchange. Since managed runtimes like Java Virtual Machines (JVMs) store data in an opaque object-based format, they have to transform objects into interpretable binary streams before exchanging. The transformation contains two phases: a *serialization* phase transforming objects into a byte array, and a *deserialization* phase transforming the byte array back into objects. The object serialization/deserialization (OSD) mechanism introduces considerable transformation overhead and has become a

significant performance bottleneck in distributed object transmission, especially for applications demanding large-scale data exchange through network [3, 6, 13, 15, 44].

Prior work has recognized the performance problem in OSD and proposed different approaches, both in software [26, 38, 39] and hardware [16, 32, 40, 46], to mitigate its effect. However, those approaches mainly focus on optimizing specific phases in OSD, and the data transformation is still inevitable. Furthermore, although they can boost the performance of OSD, many of them require extra programming efforts to annotate serialization points or change the original inter-JVM communication model. Last but not least, they treat the transferred data as a monolithic byte array instead of individual objects, which makes it difficult to identify duplicated transmission and misses optimization opportunities.

Instead of optimizing OSD, this work aims at directly eliminating the whole OSD process. To this end, this work proposes Zero-Change Object Transmission (ZCOT), which provides an ideal data exchange mechanism where objects are transferred among JVMs through **direct object copying**. When a JVM receives objects from others, it can directly process them without any modifications (*Zero-Change*). ZCOT removes the demands for object transformation and thus improves the performance of data exchange.

However, it is non-trivial to achieve zero-change communication given each JVM manages objects in a process-specific and opaque format. To this end, ZCOT first introduces a globally shared abstraction named *exchange space*, a part of the Java heap space accessible for multiple JVMs in a distributed environment. ZCOT further adopts its distributed class-data sharing (DCDS) mechanism, which provides a unified object format to make objects in the exchange space interpretable for all JVMs. To remain compatible with traditional OSD-based applications, ZCOT proposes a two-level transmission mechanism to bridge the gap between object-based copying and traditional byte-based transmission.

As ZCOT introduces a globally shared exchange space, it is responsible to manage objects shared among multiple JVMs. By introducing a metadata server, ZCOT memorizes

the stored location for objects and helps build data transmission channels between JVMs. Since objects in big-data analytics are usually exchanged as a whole dataset, ZCOT embraces *group-based object management*, which organizes objects in groups and greatly reduces the traffic between the metadata server and JVMs. Furthermore, ZCOT also integrates with the garbage collections (GC) triggered in individual JVMs and reduces the GC pause time.

ZCOT sends objects instead of byte arrays during transmission, which makes it object-conscious and easier to identify duplicated objects. This work thus proposes a data deduplication mechanism to further optimize the data transmission. The deduplication module in ZCOT leverages the exchange space abstraction to memorize which objects have been sent and avoids unnecessary object transmission in the future. Nevertheless, deduplication may introduce references (or dependencies) among different datasets. To this end, ZCOT extends its distributed memory management module to consider inter-group dependencies.

This work implements ZCOT in the HotSpot JVM of OpenJDK 11, the long-time-support version for OpenJDK. ZCOT is well-integrated with existing features in OpenJDK (like APPCDS [30]) to remain friendly to Java developers. We evaluate ZCOT against state-of-the-art OSD libraries and optimizations with both the micro-benchmark and macro-benchmark. The micro-benchmark contains both basic and complicated data structures for data transmission, while the macro-benchmark contains two big-data analytics frameworks (Spark and Flink). The result for micro-benchmark shows that ZCOT outperforms other OSD libraries, especially for complicated data structures, and reaches up to $4.35 \times$ speedup compared with Naos [39], a state-of-the-art optimization on OSD. As for macrobenchmark, ZCOT outperforms the default OSD libraries in both Spark and Flink and thus boosts the application time by up to 23.6% and 22.2%, respectively.

To summarize, the contribution of ZCOT includes:

- A distributed shared abstraction named *exchange space* to enable zero-change object transmission among JVMs while remaining compatible with traditional OSD-based applications.
- A memory management mechanism on the globally shared space integrated with GC in individual JVMs.
- A data deduplication module to identify and eliminate unnecessary object transmission for further performance improvement.
- Experiments on communication-intensive workload to show the performance improvement of ZCOT over existing OSD libraries.

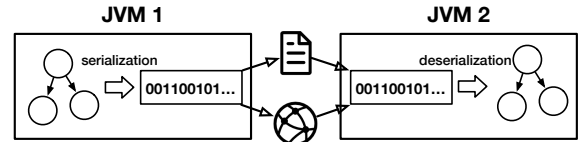


Figure 1: The workflow of OSD

2 OSD Background

2.1 OSD

Language runtimes provide a high-level abstraction for platform-independent code execution. As for user objects, runtimes store them with an opaque format, which maintains object data together with corresponding metadata (type information, synchronization, memory management, etc.). Taking Java as an example, JVMs maintain a *header* for each object to store its metadata.

However, when data exchange among JVMs is required, objects must go beyond the runtime scope. For example, objects might be persisted into disks and reused by other JVMs later; they can also be sent and received through network. To support those scenarios, objects have to be interpretable even when leaving JVMs. Therefore, JVMs embrace the object serialization/deserialization (OSD) mechanism, which transforms Java objects into a generalized data format (serialization) and transforms back when reusing in JVMs (deserialization). The Java system library (JSL) already provides a built-in OSD library for applications. Figure 1 shows the workflow of JSL’s OSD. As for the serialization part, objects are transformed into a byte array that follows a data format agreed among JVMs. The byte array will be written into disks or sent through network. When another JVM receives the byte array, it transforms the byte array back into objects through deserialization.

The OSD mechanism has two major advantages. First, the library provides a general-purpose data format so that Java objects can be transformed among JVMs with different versions and configurations. Second, the serialized data is compressed and induces smaller footprints in both disks and network.

2.2 Limitations and opportunities

The major disadvantage for OSD is its performance penalty. The performance problem of OSD in big-data analytics is three-fold.

Transformation overhead. OSD introduces extra phases for object persistence and transmission. To serialize an object, OSD should traverse all its reachable objects and store their type information. As for deserialization, OSD should scan serialized data and reconstruct objects.

Memory footprint. OSD generates a considerable number of temporary objects during data transformation. As shown

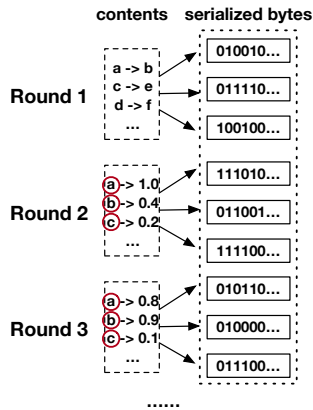


Figure 2: Duplicated data transmission in the page-rank application

in Figure 1, byte arrays are generated during serialization and become useless after sending out. Those temporary objects can increase the memory pressure and cause more frequent GCs.

Duplicated transmission. Big-data analytics leverage OSDs in many rounds of communication and duplicated objects may be repetitively transformed and exchanged in each round. Figure 2 shows a concrete example in Spark [44], which calculates the popularity for each URL (simplified as letters) with the page-rank algorithm [31]. Since the algorithm executes for multiple iterations, the data transmission is also conducted in many rounds. In the first round, the URL-based network topology is sent through network, which consists of many string pairs to indicate the point-to relationships among URLs. In the later rounds, the rank value for each URL is iteratively propagated, which is organized as key-value pairs. Note that the strings in the key-value pairs are URLs that have been sent in the first round. Unfortunately, since all objects have been transformed and merged into byte arrays, JVMs cannot tell that some objects have been received before. They have to receive all objects as a monolithic byte array, which leads to unnecessary network transmission and OSD phases. In the Spark page-rank application, over 60% of transferred objects are duplicated.

Furthermore, the advantages of OSD also fade with advances in hardware technologies. For example, the bandwidth of off-the-shelf network devices can reach 100Gb/s or larger, which makes network transmission time less important, so OSD may become a more significant bottleneck. On the other hand, the general data format is not always required. Therefore, many optimizations have been proposed to reduce the performance overhead of OSD, both in software [26,38,39] and hardware [16,32,40,46]. Since hardware-based approaches require building customized hardware accelerators to improve OSD, this work mainly focuses on software-based approaches with off-the-shelf hardware.

2.3 State-of-the-art optimizations

The basic idea behind OSD is to achieve an *agreement* on object representation among JVMs. Therefore, optimizations on OSD should consider how to create the agreement so that objects can cross JVMs' boundaries. Besides, they also need to consider issues like compatibility with existing applications.

Kryo. Kryo [38] is a fast OSD library for Java. Compared to JSL's OSD, Kryo refines the binary data format to achieve a smaller serialized data size and better performance. Applications like Spark have leveraged Kryo as its default serializer. Nevertheless, Kryo does not eliminate any phases in OSD; objects still need to be transformed back and forth.

Skyway. Skyway [26] proposes to directly send object graphs instead of serialized bytes. With Skyway, the serialization phase is nearly removed as objects are no longer transformed to a binary format. Although Skyway has simplified phases in OSD, modifications on objects are still required. First, it needs to transform the type information in the headers to a globally-agreed ID so that it can be identified by all JVMs. Second, it needs to fix references after copying, as objects have been moved to different addresses. Moreover, Skyway also requires programmers to mark the point where the serialization phase starts manually.

Naos. Naos [39] is a network-specific data transmission mechanism. Similar to Skyway, Naos also employs a global service to reach agreements for types, but it relies on RDMA technology to achieve rapid zero-copy object transmission. However, Naos still requires modifications on both object headers and references. Besides, it only supports network-based transmission, and existing applications need significant modifications to leverage Naos.

2.4 Summary

Prior optimizations have proposed different solutions to reduce the overhead of OSD. However, they cannot eliminate the whole OSD process. Table 1 compares the built-in OSD in JSL with other optimizations. Although recent work like Naos eliminates the serialization phase, a deserialization phase is still required to fix the type information and the references. Besides, none of them has considered the duplicated data transmission problem.

This work proposes ZCOT (short for Zero-Change Object Transmission), which aims to eliminate the whole OSD process during data exchange. In ZCOT, object transmission is conducted in the most straightforward way: the sender JVM copies objects and the receiver can directly use them without any modifications. ZCOT also considers the duplicated transmission problem and provides a deduplication module. Finally, ZCOT is not bound to specific network technologies (like RDMA) and provides easy-to-integrate interfaces for existing applications.

Data transmission mechanisms	Serialization	Deserialization	Ease of Integration	Data deduplication
JSL	Slow	Slow	Yes	No
Kryo	Medium	Medium	Yes	No
Skyway	Fast (removed)	Medium	Medium	No
Naos	Fast (removed)	Medium	No	No
ZCOT (this work)	Fast (removed)	Fast (removed)	Yes	Yes

Table 1: Comparisons on existing OSD optimizations against our work ZCOT

3 Design of ZCOT

3.1 Overview

The core idea of ZCOT is to build a distributed-shared-memory (DSM)-like abstraction for JVMs running on different machines. Figure 3 illustrates the architecture of ZCOT. In a ZCOT-enabled system, the heap for each JVM consists of two parts: its private space (the original Java heap) and a globally shared *exchange space*. Objects are originally managed in the private space. When they require to be sent through network or persisted into disks, they will be copied to the exchange space. The exchange space is an abstraction available for all JVMs; each JVM can directly access objects therein. Therefore, object transmission can be achieved with direct copying to the exchange space, and the whole OSD process can be eliminated.

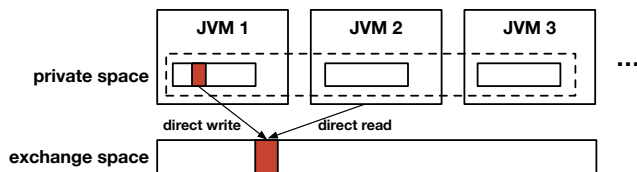


Figure 3: The architecture of ZCOT

The idea for building a DSM-like abstraction is well-known and has been studied for decades [2, 7, 9, 14, 20, 21, 25, 33–35, 41, 42, 45]. Although our exchange space shares similar wisdom with DSM, it is only used for data exchange and does not need to tackle complicated issues like coherence. It also assumes objects in the exchange space are immutable, which usually holds for big-data analytics like Spark and Flink. If a write operation occurs on objects in the exchange space, ZCOT creates a copy for it on the JVM’s private heap. Nevertheless, combining the DSM concept with data transmission in high-level languages is still not trivial. To enable efficient and easy-to-use object transmission, ZCOT has to resolve the following challenges.

- How to build a shared exchange space so that all JVMs can access it freely? (Section 3.2)
- How to leverage the exchange space abstraction to support OSD-based applications? (Section 3.3)
- How to manage objects in the exchange space in the presence of garbage collections in individual JVMs?

(Section 4)

- How to resolve the duplicated transmission problem? (Section 5)

3.2 Distributed class-data sharing

ZCOT relies on its distributed class-data sharing (DCDS) mechanism to build a globally accessible shared space. DCDS guarantees that class-related metadata will be mapped into the same virtual memory address for all JVMs. This helps JVMs to achieve an agreement on the class metadata, so no type-related modifications (e.g., identifiers) are required.

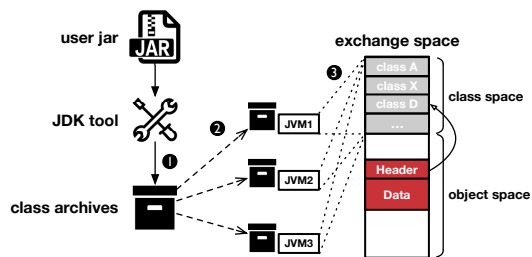


Figure 4: The workflow of distributed class-data sharing

Figure 4 elaborates the workflow of DCDS. First, the cluster manager should prepare a shared class archive for all JVMs. The class archive should contain all classes whose corresponding object instances would be shared during inter-JVM communication. ZCOT relies on the tools provided by OpenJDK to generate such class archives [30]. Afterward, the archive will be used during JVM startup, and classes in the archive will be mapped to a given virtual address. The virtual address range is also memorized and marked as a part of the exchange space (*class space* in Figure 4). This step assures that JVMs share the same view on the classes. As shown in Figure 4, an object in the exchange space stores a reference to its class-related metadata. Since the reference points to the class space, the object’s class information is interpretable for all JVMs. Although DCDS requires the data types of applications should be known in advance, mainstream big-data analytics frameworks usually guarantee this by sending a fat jar file for execution.

Figure 5 shows how ZCOT transfers objects through network with its DCDS support. First, the sender JVM applies for an available memory chunk in the exchange space for object copying. This is achieved by communicating with an external

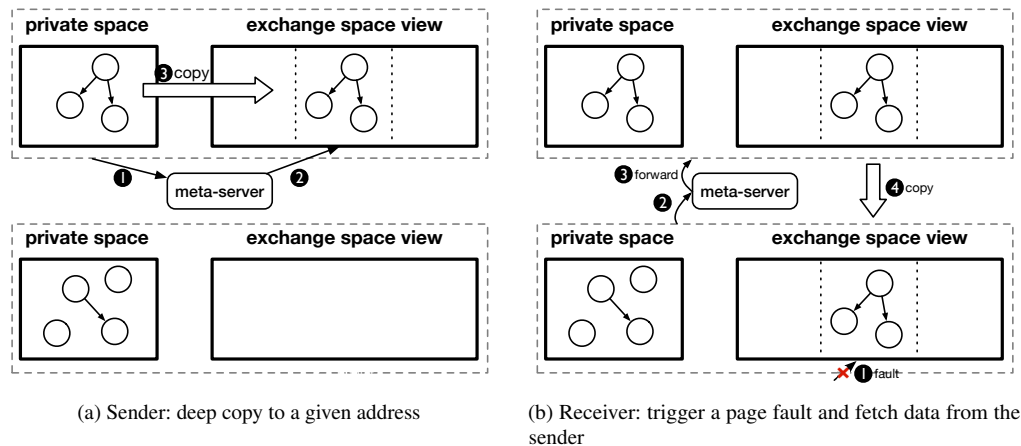


Figure 5: The workflow of ZCOT

metadata server (details in Section 4). Second, the sender JVM copies objects to the chunk’s memory address. This step is similar to a deep copy in a normal Java application. To detect cycles and avoid repeated copying on the same object, we add a marker word in each object header to store its new address if it has been copied.

The copied objects are kept on the sender machine and lazily retrieved by receivers. When a receiver JVM tries to access this part of data (Figure 5b), it encounters a page fault since the data is unavailable on its machine. We have registered the page fault handler in ZCOT-enabled JVMs so that they can request the metadata server for faulted pages. The metadata server has tracked the ownership of memory addresses in the exchange space, so it forwards the request to the data owner. Afterward, the sender builds a connection with the receiver and puts the requested objects to the desired address. Now the receiver can directly access those objects for further processing, with neither metadata updating nor reference fixing (namely *zero-change*).

3.3 Supporting OSD-based scenarios

Thanks to the exchange space abstraction, a JVM can directly access received objects without any modifications. However, this mechanism is not compatible with traditional OSD-based applications, which usually adopt byte arrays for communication. To this end, ZCOT should provide user-friendly interfaces to integrate easily with applications.

Programming interfaces. JSL provides stream-based classes for OSD implementation. The `ObjectOutputStream` class provides the `writeObject` method to serialize an object into a stream (usually files or network). Similarly, the `ObjectInputStream` class provides the `readObject` method to deserialize data into objects. Therefore, prior OSD optimizations like Skyway implement new serializers/deserializers by inheriting those two classes for

ease of integration. ZCOT adopts a similar strategy and Figure 6 shows its basic classes: `ZCObjectOutputStream` and `ZCObjectInputStream`, which are subclasses of `ObjectOutputStream` and `ObjectInputStream`, respectively. Compared with `ObjectOutputStream`, `ZCObjectOutputStream` slightly modifies the interface for `writeObject` to support different OSD-based scenarios (discussed later). To use ZCOT-based communications, applications only need to replace the original stream classes with ours. In contrast, prior work requires developers to modify the original communication model or annotate the serialization points [26, 39].

OSD-compatibility. To remain compatible with OSD interfaces (`writeObject` and `readObject`), ZCOT should also transfer data with byte arrays. To this end, ZCOT adopts a two-level transmission mechanism. As illustrated in Figure 7, ZCOT transfers data via both *frontend* and *backend*. The frontend transmission is compatible with OSD interfaces, but it only sends metadata, including the object’s start address and the data length. When `ZCObjectInputStream` receives the metadata through `readObject`, it directly accesses the corresponding address and fetches objects through backend transmission if a page fault is triggered (as mentioned in Figure 5b). ZCOT will launch dedicated VM threads in both sender and receiver JVMs to transfer the requested objects. This two-level design fills the gap between the byte-based OSD interfaces and the object-based transmission in ZCOT.

Supporting different OSD scenarios. In OSD libraries, objects are serialized and written into a stream (e.g., the `out` variable defined on Line 3 in Figure 6) when invoking `writeObject`, which are usually redirected into files or network. To support both scenarios, ZCOT adds a parameter `volatile` in the constructor of `ZCObjectOutputStream` (Line 6). When `volatile` is set to `false`, the copied objects will be written into a file, and the memory pages can be soon reclaimed

```

1 // Output class
2 class ZCObjectOutputStream extends ObjectOutputStream {
3     private OutputStream out; // Private output stream
4
5     // Constructor
6     public ZCObjectOutputStream(OutputStream out,
7         boolean volatile /* Mode */)
8         throws IOException {...}
9
10    // Compatible with the serialization interface
11    public void writeObject(Object obj)
12        throws IOException {...}
13    ...
14 }
15 // Input class
16 class ZCObjectInputStream extends ObjectInputStream {
17     private InputStream in; // Private input stream
18
19     // Constructor
20     public ZCObjectInputStream(InputStream in)
21         throws IOException {...}
22
23     // Compatible with the deserialization interface
24     public Object readObject()
25         throws IOException {...}
26     ...
27 }

```

Figure 6: Basic classes in ZCOT

through GC (details in Section 4). Nevertheless, those objects still reserve a corresponding virtual address in the exchange space. When the object data is read by other JVMs, the metadata server asks the sender to pass the file so the receiver can map it to the corresponding memory address. The case is simpler when *volatile* is *true*, which indicates a network-based transmission. In this scenario, objects are only kept in memory and can be reclaimed only if they have been read by others.

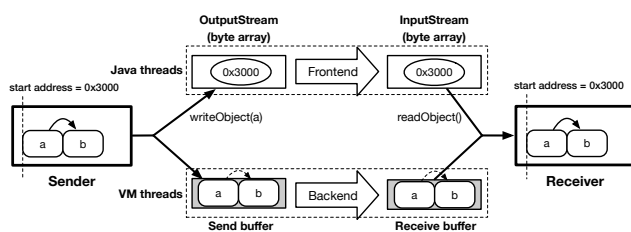


Figure 7: The two-level data transmission mechanism in ZCOT

Assumptions. Note that ZCOT is mainly designed to improve the data exchange phase for big data analytics, so it makes several assumptions about the transferred data. First, all classes related to communication should be known in advance so that they can be packed into the class archive. Second, the transferred objects are read-mostly, otherwise copy-on-write

operations would be triggered for modification operations. Lastly, objects are managed in large groups and share similar life cycles, so they can be efficiently managed in the exchange space. Since representative big-data analytics systems like Spark conform to the above assumptions, ZCOT works well for them.

4 Memory Management

Since the global exchange space is built atop a DSM-like abstraction, ZCOT should manage objects distributed to different machines. Furthermore, the managed runtimes complicate the scenario as they introduce their own memory management strategy: garbage collections (GC). This section will introduce how ZCOT manages the distributed exchange space while remaining harmonized with GC in JVMs.

4.1 Group-based management

Unlike traditional DSM-based systems, ZCOT introduces *group*, a semantic-aware notion for distributed memory management. As analyzed in Section 2, big-data analytics frameworks treat serialized objects as a whole dataset (monolithic byte array) and retrieve them together. Therefore, ZCOT puts all objects copied in the same `writeObject` invocation to a group so that they are managed together. When a receiver encounters a page fault, ZCOT will send all related data pages belonging to the same group to the receiver and avoid future faults. This mechanism, namely *group-based prefetching*, leverages the semantics in the OSD scenario to mitigate the page-based management overhead in traditional DSM.

4.2 Metadata server

The metadata server is the core module for ZCOT’s memory management. JVMs communicate with the metadata server through remote procedure calls (RPCs) to acquire or release memory resources in the exchange space. Figure 8 illustrates the core data structures in the metadata server. The metadata server is agnostic to groups; groups are only managed by individual JVMs. It partitions the shared exchange space into equal-sized memory chunks (256MB by default) for memory allocation and deallocation. It also maintains an *allocation bitmap* to mark if a chunk has been allocated. Each chunk is assigned with an integer ID, which is calculated by its relative offset compared with the exchange space’s start address. To track the stored locations of chunks, the metadata server maintains a *copy set* for each chunk, which is stored in a *chunk mapping table*. The copy set contains JVMs storing a copy of the corresponding chunk (in memory or disk), which are also represented with integer IDs. The mapping between a JVM’s ID and information (e.g., IP address) is stored in a separated *member table*.

Since each JVM needs to communicate with the metadata server, its reliability becomes considerable. To tolerate failures on the metadata server, we can introduce replicas for it, and the overhead would be acceptable given the low frequency of communications between the metadata server and worker JVMs (several times in a data-processing stage lasting for seconds).

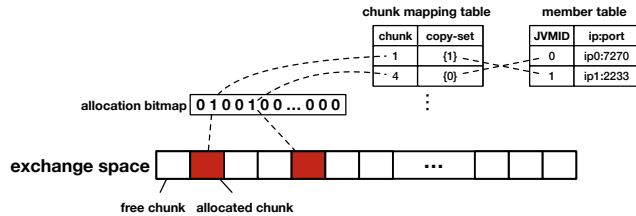


Figure 8: Important data structures in the metadata server.

4.3 RPC interfaces

The metadata server provides four important RPC interfaces listed below.

```
int register(std::string ip, int port);

Chunk* acquire();

Chunk* get_remote(Address addr);

int release(Chunk* chunk);
```

register. `register` is only invoked when a JVM is launched. ZCOT has provided a JVM option `-XX:+UseZCOT`, and a JVM enabling this option automatically spawns an RPC thread and sends a `register` RPC to the metadata server with its IP address and listening port. After receiving the RPC, the metadata server saves the IP and port number to the member table, generates an integer as the JVM's ID, and returns with the ID. For subsequent RPCs, JVMs should always attach the returned ID to help the metadata server maintain the stored locations of objects (omitted in the interfaces above).

acquire. When a JVM runs out of allocated memory from the exchange space, it should send `acquire` RPCs for more memory resources. After receiving an `acquire` request, the metadata server scans its bitmap to allocate an available chunk. Afterward, the metadata server memorizes the relationship between the allocated chunk and the JVM's ID and returns the chunk. To reduce the overhead of bitmap scanning, ZCOT memorizes the address of the last successfully allocated chunk and starts scanning there. If the scanned address reaches the end of the exchange space, ZCOT will continue scanning from the beginning. To handle simultaneous `acquire` requests, ZCOT introduces a bitmap lock to ensure the bitmap is exclusively accessed.

get_remote. The `get_remote` interface is used by JVMs encountering a page fault when accessing a virtual address. Since a page fault indicates the requested objects are not stored locally, the JVM sends `get_remote` to fetch the corresponding chunk. After receiving `get_remote`, the metadata server gets the corresponding chunk containing the address and finds which JVMs store the chunk by scanning the chunk mapping table. As illustrated in Section 3.2, the metadata server forwards the request to the corresponding JVM for actual data transmission. Since the size of a chunk is relatively large, sending chunks may introduce considerable performance overhead. To reduce the transferred data size, the sender JVM only sends used pages in the chunk, which are represented as the length of data in the frontend transmission (Figure 7). Due to ZCOT's group-based prefetching mechanism, the sender may directly send multiple chunks in the same group to the receiver. In this case, the receiver is responsible for sending an auxiliary RPC to update the copy set in the metadata server.

release. The `release` interface is relatively simple. When a JVM finds that objects in a chunk are no longer used, it sends `release` to give up this chunk. After receiving `release`, the metadata server removes the JVM's ID from the corresponding copy set in the chunk mapping table. If no JVM stores this chunk, the metadata server will reclaim it by marking the corresponding bit as *free* in the bitmap.

4.4 Garbage collection

JVMs have already implemented their garbage collection (GC) algorithms to automatically reclaim unused memory. When GC is triggered, JVMs track all live objects and reclaim memory consumed by dead ones. Since objects in the exchange space are reachable from individual JVMs, they will also be affected by GC. To this end, ZCOT has integrated its memory management strategy with G1, the default GC algorithm in OpenJDK, to ensure the correctness of distributed memory management and reduce GC overhead.

G1 basics. G1GC (short for Garbage-first Garbage Collection [8]) is the default garbage collector after OpenJDK 9 [29]. G1 divides the Java heap into equal-sized *regions* for ease of management. It also maintains per-region metadata named *remember sets* to memorize all references pointing to objects in the same region. The remember set is updated by instrumenting all write operations in Java code (also known as *write barriers*). The G1 algorithm is mostly stop-the-world, which means that application threads should be paused until GC ends. During GC¹, each selected region is processed simultaneously: a dedicated GC thread scans the remember set of a region, finds all reachable objects, and copies them to an empty region (named *survivor region*).

Integrated with G1. ZCOT extends the region-based design of G1 to support the exchange space. It proposes *ZCRe-*

¹For simplicity, we only discuss the young GC and mixed GC in G1

gion, a new kind of region allocated from the metadata server. Compared with regions in G1, the size of ZCRegion is not fixed. Each ZCRegion corresponds to a group in the exchange space, and all objects therein are expected to have the same life cycle. Since objects in ZCRegions have different behaviors compared with those in other regions, G1 should treat them specially. First, we modify the behavior of write barriers to consider ZCRegions. When a reference points to objects in a ZCRegion, we do not memorize this reference but only mark the ZCRegion as used. This is because objects in a ZCRegion are only collected when no references point to any of them. Similarly, GC threads do not need to scan ZCRegions during GC because all objects are treated as alive if there exists any reference pointing to the region. When GC ends, the JVM will scan all ZCRegions and find those containing no incoming references. For those ZCRegions, the JVM invokes the `release` RPC to reclaim corresponding chunks. If objects in a group are written into disks, the corresponding ZCRegion can also be reclaimed by GC, but the JVM does not invoke `release` since the virtual address is still reserved by the group.

In summary, our design successfully integrates the memory management of the exchange space with G1GC. When GC ends, the memory resource in the exchange space is automatically reclaimed by following the reachability-based algorithm. Furthermore, by specially handling regions in the exchange space, we avoid unnecessary metadata tracking and object scanning. In some cases, this design can even reduce GC pause time (as shown in Section 6.3).

5 Transmission Deduplication

Since ZCOT sends objects instead of byte arrays during transmission, it would be much easier to track transmitted objects and conduct deduplication. This section introduces the data deduplication module in ZCOT based on its object-centric transmission mechanism.

5.1 Overview

Figure 9 shows the effect of ZCOT's data deduplication module in the aforementioned page-rank example (compared against Figure 2). When sending the URL-based network topology in the first round, the sender has copied all URL string pairs (together with two string objects) into their corresponding addresses. In the next few rounds, the application sends key-value pairs to update rank values for each URL. Since all key-value pairs are sent as objects, it is much simpler for ZCOT to find that all URL objects have been sent. Therefore, the sender can directly update the references in those key-value pairs with the addresses in the exchange space and thus remove duplicated transmission on URL objects.

ZCOT runtime should be further extended to achieve data deduplication. First, ZCOT should track copied objects to rapidly find duplicated transmission. Second, ZCOT should

manage dependencies among object groups for safe memory reclamation.

5.2 Duplication detection

A straw-man design for duplication detection would be scanning all objects in the exchange space. However, this design would induce considerable overhead given the large number of objects. ZCOT instead follows a simple detection criterion: if an object is in the exchange space, an attempt to copy it is a duplicated one.

We still use page-rank as an example to explain ZCOT's duplication detection. Suppose a JVM receives the network topology in round 1 (consider Figure 9); it reads URL objects from the exchange space and uses them in the following rounds. Therefore, when it propagates updated rank values to other JVMs, it still uses the URL objects received from others. When copying the URL-rank pairs in the next few rounds, ZCOT checks each object's address and thus avoids copying those URL objects already in the exchange space.

5.3 Dependency management

Although data deduplication in ZCOT can reduce the network overhead by avoiding repeated copying on the same object, it also complicates memory management by introducing inter-group references. As mentioned in Section 4.1, each invocation to `writeObject` creates a new group for object management, and each group is separately used by calling `readObject`. After deduplicating objects from different groups, objects in a group can hold references to those in another group, which should be correctly handled especially when a group is being garbage collected. To this end, ZCOT has managed those references as *dependencies* among groups.

Due to the large number of inter-group references, ZCOT does not maintain reference-level dependencies. When a group holds a reference to any objects in other groups, ZCOT marks the group as dependent on others. The dependency tracking is still achieved by extending the write barriers. To memorize all dependencies, ZCOT extends the chunk mapping table in the metadata server to contain a *dependency set* for each chunk, which stores all other chunks it relies on. When a JVM finds that its group relies on another group after deduplication, it sends a new RPC `add_dependency` to the metadata server. Since the metadata server is not aware of groups, the RPC should specify all chunk IDs owned by the group it relies on. Those chunk IDs will be added to the corresponding dependency set by the metadata server.

Figure 10 uses an example to illustrate how ZCOT leverages dependencies during object copying. After encountering a page fault on chunk 4, the receiver JVM sends a `get_remote` RPC to the metadata server. By fetching the dependency set in the chunk mapping table, the metadata server finds all chunks that chunk 4 depends on. Afterward,

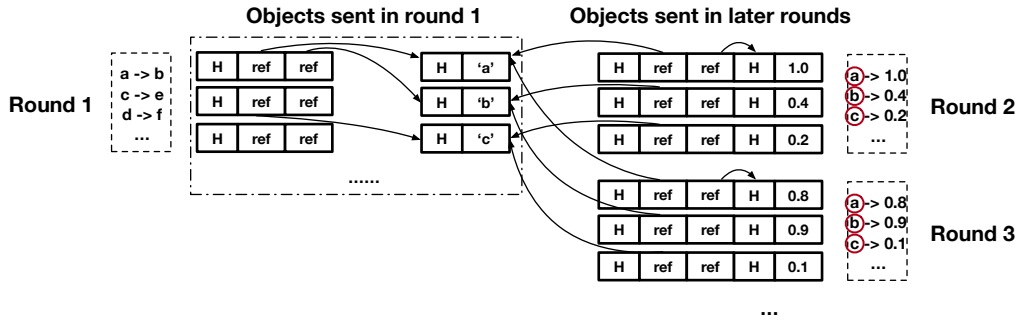


Figure 9: ZCOT avoids duplicated object transmission in page-rank

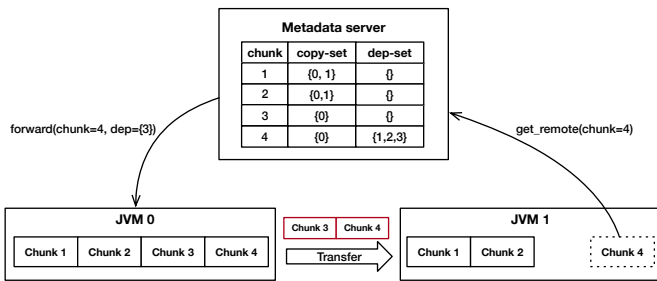


Figure 10: ZCOT avoids sending duplicated data with dependency tracking

ZCOT checks the copy set for each chunk to find if the receiver JVM already has a copy. If the receiver does not have a copy, ZCOT adds the corresponding chunk ID in a message, which will be forwarded to the sender JVM later for real data transmission. In our example, since the receiver JVM already has copied chunk 0 and 1, it only needs to receive chunk 4 (requested) and chunk 3 (dependent). This example indicates that ZCOT can avoid duplicated data submission with slight modifications on the metadata server.

5.4 Garbage collection

Adding dependencies also complicates GC for individual JVMs. Since a group (represented as ZCRegions) can be referenced by others stored in remote JVMs, local GC cannot determine if a group can be safely reclaimed. For example, suppose JVM 0 stores a group (chunk 0) that contains a reference to another group (chunk 1) stored on JVM 1. Although JVM 1 no longer contains references to chunk 1, the chunk should not be collected because JVM 0 may access it through references in chunk 0. To this end, we extend G1GC to consider remote inter-group references.

In our refined GC algorithm, once a JVM detects a ZCRegion has incoming references from other ZCRegions (through write barriers), it marks the region as *pinned* and thus cannot be reclaimed. It also sends the dependency relationship to the metadata server through RPCs. When GC ends, the JVM

skips all pinned ZCRegions and only collects those with no incoming references. A pinned ZCRegion can be reclaimed when the metadata server finds that all chunks relying on it have been released. In this case, the metadata server will send a `canRelease` message to all JVMs in the corresponding copy set, and those JVMs will mark the ZCRegion as *unpinned* to safely reclaim it in later GC cycles.

5.5 Internalization

Big-data analytics usually generate a large number of objects with simple types, such as Integer, String, Double, etc. OpenJDK has provided an *internalization* mechanism to merge those objects with the same content together. For example, Integer objects whose values are between -128 and 127 would be merged into one if their values are equal. ZCOT also embraces this mechanism for deduplication, but in its distributed exchange space. It extends DCDS so that all JVMs allocate a small region at the same virtual address during start-up to contain globally-shared Integer objects. Thanks to this optimization, the number of transferred Integers can be greatly reduced.

6 Evaluation

6.1 Experimental setup

ZCOT is implemented atop the HotSpot JVM in OpenJDK 11.0.8-GA, with 8,327 lines of C code and 654 lines of Java code. We leverage the following workloads to evaluate ZCOT.

Microbenchmark. The microbenchmark contains four different data types used in prior work [26, 39]: 2-dimension points, key-value pairs, hashmaps, and media objects. To simulate big-data scenarios, we transfer them in large arrays whose length is 65536. Since some baselines crashed for large arrays of media objects, we reduced the length to 16384 for this data structure.

Spark. Spark (v3.0.0) is a data analytics engine that requires massive data transmission among JVMs.

Flink. Apache Flink [6] (v1.14) is a distributed data processing engine for both batch and streaming workloads.

As for baselines, we compare ZCOT with two commonly-used OSD libraries (JSL and Kryo) and two state-of-the-art OSD optimizations (Naos and Skyway²).

Our test environment includes a cluster with four nodes connected by 100 Gbit/s Mellanox ConnectX-5 NICs. Each node contains dual Xeon E5-2650 CPUs and 128GB DRAM.

6.2 Microbenchmark

To directly compare ZCOT with state-of-the-art OSD optimizations, we leverage the *microperf* tester in the Naos’ open-source repository for evaluation. The tester involves a sender and a receiver deployed on two separate machines and reports the communication time with different type of data objects. The heap size for all workloads is 16GB.

Figure 11 shows the results for ZCOT and other baselines, which are the average of 1000 times of repetitive execution. ZCOT achieves the best performance of all except for 2-dimensional points. The average speedup is 2.28×, 1.94×, 2.19×, 3.95× compared with Naos, Skyway, Kryo, and JSL, respectively. The result also suggests that ZCOT performs better for complicated data structures. The media class from the Java serialization benchmark set (JSBS) [37] is the most complicated one, so the improvement is the largest especially against Naos (4.35×). This is because the computation overhead increases when the data structure becomes more complex. For simple data structures like points, ZCOT’s reduction on data transformation is offset by larger network overhead, so it performs slightly worse than Naos and Skyway.

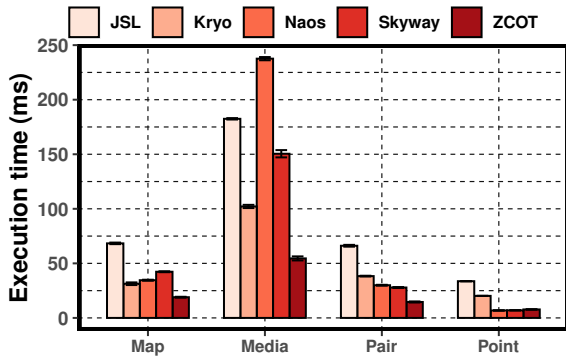


Figure 11: The evaluation results for microbenchmark

6.3 Spark

Ease of integration. To adopt ZCOT in Spark, we need to implement a new data serializer `ZCSerializer` to replace the default `KryoSerializer`. Although the name seems to

²Both implemented by Naos’ authors

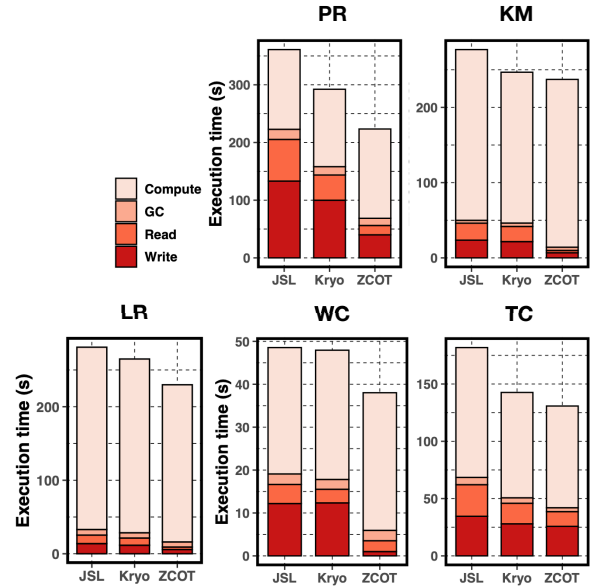


Figure 12: The performance of Spark applications

involve OSD phases, it is only for compatibility considerations and still remains zero-change during transmission. `ZCSerializer` contains 70 lines of code, and most of them are inherited from the JSL serializer. Furthermore, we replace the original stream classes from JSL with ours. If a Spark user wants to enable ZCOT, she only needs to (1) configure the `spark.serializer` to `ZCSerializer` and (2) add `-XX:+UseZCOT` to the launch option of all JVMs, which is quite simple.

Evaluation results. We leverage five applications in the example directory of Spark for evaluation. Their descriptions and evaluated datasets are shown in Table 2. We configure one node as the metadata server and Spark master while the other three servers as Spark workers. The Java heap size for each node is set to 80GB.

Figure 12 shows the results for all applications. The results indicate that ZCOT can improve the performance by 13.9% and 24.1% on average compared with Kryo and JSL, respectively. Although Kryo has optimized the OSD performance over JSL, our evaluation shows that the data transmission can be further improved.

Application	Dataset
PageRank (PR)	LiveJournal [4]
Word Count (WC)	LiveJournal
KMeans (KM)	USCensus1990 [10]
Transitive Closure (TC)	Blogs [1, 17]
Logistic Regression (LR)	SUSY [5]

Table 2: Evaluated applications and datasets for Spark

We have further broken the results into four different phases: write (serialization), read (deserialization), compu-

tation, and garbage collection (GC). Since the four phases are not overlapped in Spark (the GC phase only contains stop-the-world time), the accumulated time is equal to the overall execution time. Figure 12 indicates that the performance mainly comes from the improvement in OSD-related parts. Since OSD occupies a considerable portion in page-rank execution, ZCOT can reach its best improvement (23.6% and 38.1% w.r.t. Kryo and JSL). Averaged across all applications, ZCOT can reach $4.19\times$ speedup in the write part and $2.95\times$ in the read part over the default Kryo serializer ($4.52\times$ and $3.81\times$ speedup for write and read part in JSL). As for GC, ZCOT shows comparable pause time with others. In PR, LR, and TC, the GC time is even shorter than JSL and Kryo. Although ZCOT needs to manage the copied groups (ZCRegions), its coarse-grained collection strategy avoids scanning objects inside ZCRegions. Moreover, ZCOT avoids generating monolithic byte arrays by eliminating the serialization phase, which can mitigate the memory pressure and introduce less frequent GC.

Note that the computation time in ZCOT is somewhat larger than that in JSL and Kryo. This can be explained by two reasons. First, since ZCOT does not compress the object contents during transmission to achieve zero-change, the transferred data size is larger than JSL and Kryo, which leads to larger network overhead (included in the computation part). Second, the data deduplication module makes objects in the same dataset scattered into different virtual address ranges, which may lead to more random memory accesses and cache misses. Nevertheless, the overall performance improvement is satisfying.

Results for deduplication. We have also studied the effect of our data deduplication module. As shown in Table 3, ZCOT can reduce the transferred data size for all four applications, ranging from 8.1% to 53.8%. Even for the non-iterative application (WC), ZCOT is also helpful thanks to its internalization optimization technique. Meanwhile, LR and KM receive smaller savings because they generate many different Double objects in each iteration, which cannot be reused and deduplicated. The result indicates that duplicated transmission is common in data analytics and ZCOT’s optimizations are helpful. Note that the number of transferred bytes after deduplication is still much larger than that in Kryo and JSL, since both of them converts objects in a compact format before transmission. Therefore, it is still preferred to use ZCOT with larger network bandwidth.

	PR	WC	TC	KM	LR
dedup	15.25	4.13	5.03	5.37	5.55
no-dedup	31.64	5.50	10.88	5.86	6.04

Table 3: Average transferred bytes (GB) for Spark executors

Various settings and overhead analysis. We evaluate the performance of ZCOT with various settings on the heap size and the chunk size by using PR as an example. The results in

Figure 13 show that ZCOT is not sensitive to different settings and reaches similar performance. We have also studied the overhead of write barriers by running Spark applications atop ZCOT’s JVMs (with Kryo serializers) and comparing the performance against vanilla JVMs. The average overhead among all applications is 2.73%, which is much smaller compared with the improvement brought by ZCOT. We also find the average communication overhead with the metadata server is only several milliseconds for each data-processing iteration, which usually lasts for seconds.

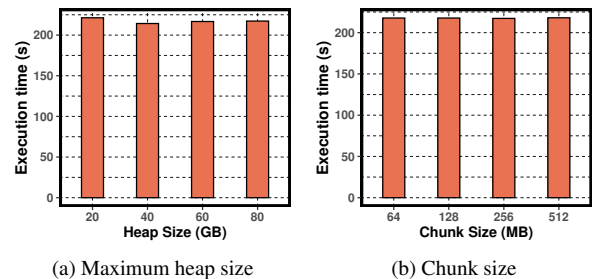


Figure 13: Results for PR under various settings

6.4 Flink

Ease of integration. We have also integrated ZCOT to Flink, another big-data analytics framework. Although Flink adopts its built-in serializer and deserializer for OSD, the integration is not complicated since we only need to replace them with ZCOT’s OSD-compatible interfaces and streams.

Evaluation results. We leverage four representative SQL queries in the TPC-H benchmark for evaluation (Q1, Q3, Q6, and Q10) and rely on its built-in generator to create input data (10GB). The configuration is similar to Spark: we launch three workers on different machines for evaluation, but the Java heap for each node is 20GB. Since the read and write phases are overlapped in Flink, we do not break the execution time into parts. The results in Figure 14 show that ZCOT outperforms the built-in serializer in Flink for three out of four queries and leads to 2.3%-22.2% improvement in query execution time. ZCOT does not improve Q6 since it does not involve a reduce operator and the amount of transferred data is limited. It performs the best for Q10 (22.2%) since it reaches $4.40\times$ improvement for the write part and $1.44\times$ for the read part. The speedup is smaller compared with Spark since Flink’s built-in serializers are manually optimized for specific data structures (like tuples). Nevertheless, ZCOT still shows better performance than the vanilla version of Flink, which suggests the importance of zero-change transmission mechanism.

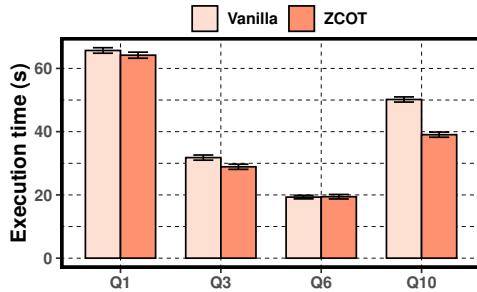


Figure 14: The performance of Flink applications

7 Related work

7.1 OSD optimizations

OSD has become a considerable performance bottleneck especially for large-scale communication-intensive applications. To optimize the time-consuming phases in OSD, prior work such as Kryo [38], Skyway [26], and Naos [39] has refined the transmission data format or leveraged the advances in network hardware technologies. ZCOT instead aims at eliminating the whole OSD process. Apart from software-based techniques, another line of work adopts hardware-based approaches to reduce OSD overhead. Optimus Prime [32] builds a data transformation accelerator (DTA) to improve the OSD throughput for microservices. Cereal [16] co-designs the data transmission format with hardware accelerators to improve the performance and energy efficiency of Spark applications. Morpheus [40] moves the deserialization phase into smart SSDs, while Hgum [46] leverages FPGAs to handle OSD tasks. ZCOT is based on off-the-shelf hardware and thus orthogonal to those hardware-based optimizations.

7.2 Distributed language runtimes

The idea for building a distributed language runtime (e.g., distributed JVMs) has been explored for decades. Java/DSM [43] builds a distributed JVM atop DSM for heterogeneous computing. JESSICA [21, 47] provides a single global thread space and transparently migrates Java threads for load balance. Comet [14] builds a DSM-abstraction for JVMs running on both mobile devices and the cloud and relies on its memory model to achieve effective code offloading. Semeru [41] proposes a universal Java heap abstraction so that a Java application can freely access all memory resources in a memory-disaggregated architecture. Those systems leverage a shared heap to synchronize data among different endpoints, but they do not consider the performance overhead of inter-JVM communication for large applications. XMem [42] enables efficient type-safe object sharing among multiple JVMs on the same physical machine, but it does not consider distributed environments. ZCOT also proposes a distributed runtime design,

but it mainly focuses on boosting data transmission among multiple JVMs.

7.3 Runtime optimizations for Java

High-level languages like Java are intensively used in large-scale, distributed applications, which stimulates research interests in runtime optimizations for performance improvement. ITask [11] makes data processing tasks interruptible when facing large memory pressure, which leads to better performance and fewer out-of-memory errors. Yak [27] divides the application execution into epochs and triggers GC when an epoch ends. Broom [12] embraces a region-based design and puts objects with the same lifecycle into the same region for fast reclamation. ScissorGC [18, 19] proposes shadow regions to improve the scalability of full GC phase. Taurus [22, 23] coordinates GC from different JVMs to reach better performance or smaller tail latency. Facade [28] and Deca [36] store massive data objects in off-heap memory to reduce GC pressure, while Gerenuk [24] enables speculative execution on serialized data to reduce both memory footprint and GC overhead. ZCOT focuses on eliminating the OSD process and duplicated object transmission, and it also collects objects by coordinating with the metadata server.

8 Conclusion

This work introduces ZCOT, which aims to eliminate the object serialization/deserialization phase in data exchange among language runtimes (like JVMs). ZCOT provides an exchange space where objects are interpretable for all JVMs, which removes the need for any data transformation during object transmission. It also uncovers the duplicated object transmission problem and provides a corresponding deduplication mechanism. The evaluation shows that ZCOT can significantly improve the performance of object transmission.

9 Acknowledgement

We sincerely thank our anonymous shepherd and reviewers for their insightful suggestions. This work is supported in part by the National Natural Science Foundation of China (No. 62172272, 61925206, 62132014). Binyu Zang (byzang@sjtu.edu.cn) is the corresponding author.

A Artifact Appendix

Abstract

ZCOT, or Zero-Change Object Transmission, is proposed to optimize data exchange among multiple Java virtual machines (JVMs) in a distributed environment. Instead of sending and

receiving data with the costly object serialization/deserialization (OSD) phase, ZCOT allows JVMs to directly communicate with Java objects, which significantly improves the data exchange time, especially for applications like big data analytics.

Scope

This artifact (including binaries, source code, documents, and scripts) is used to conduct the main experiments in ZCOT, which consists of the following two parts:

- **Micro-benchmark performance.** The result should show that ZCOT outperforms recent OSD optimizations (Skyway [26] and Naos [39]) and state-of-the-art OSD libraries (Kryo [38] and JSL) for most data structures used in Naos' microbenchmark.
- **Spark performance.** The result should show that ZCOT outperforms Kryo and JSL-based Spark applications in both data exchange and task execution.

Note that we only report numbers evaluated on our machines, so the results might be different with various hardware configurations.

Contents

We pack all related files into a zipped one, which contains the following contents.

- **README.** A file containing instructions for artifact evaluation.
- **ZCOT-jdk.** The source code of a modified OpenJDK to support ZCOT.
- **Meta-server.** The source code of the metadata server used in ZCOT.
- **Micro.** Scripts and jars used for the micro-benchmark.
- **Spark.** Since the code size of Spark is quite large, we provide an executable binary for Spark, which is slightly modified to evaluate ZCOT.
- **Naos-jdk.** A slightly modified version of Naos' OpenJDK to compare with ZCOT.

Hosting

Currently our code is not ready for open-source. Nevertheless, you can contact us via mingyuwu@sjtu.edu.cn to obtain the artifact.

Requirements

Hardware requirements. We evaluate ZCOT on four nodes connected by 100 Gbit/s Mellanox ConnectX-5 NICs. The NIC bandwidth has a significant impact on ZCOT's performance.

Software requirements. The operating system used in our machines is Ubuntu 16.04.2, but higher versions are also acceptable. Note that huge pages should be enabled to run ZCOT. Dependencies for installing OpenJDK have been listed in the README file.

References

- [1] Lada A Adamic and Natalie Glance. The Political Blogosphere and the 2004 US Election: Divided they Blog. In *Proceedings of the 3rd International Workshop on Link Discovery*, pages 36–43. ACM, 2005.
- [2] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.
- [3] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.
- [4] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54, 2006.
- [5] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5(1):1–9, 2014.
- [6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.
- [7] John B Carter, John K Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 152–164, 1991.
- [8] David Detlefs, Christine H. Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*, pages 37–48. ACM, 2004.
- [9] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 401–414, 2014.
- [10] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.
- [11] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 394–409, 2015.
- [12] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

- [13] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 599–613, 2014.
- [14] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. {COMET}: Code offload by migrating execution transparently. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 93–106, 2012.
- [15] Apache Hadoop. Hadoop, 2009.
- [16] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W Lee. A specialized architecture for object serialization with applications to big data analytics. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 322–334. IEEE, 2020.
- [17] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Statistical properties of community structure in large social and information networks. In *Proc. Int. World Wide Web Conf.*, pages 695–704, 2008.
- [18] Haoyu Li, Mingyu Wu, and Haibo Chen. Analysis and optimizations of java full garbage collection. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, pages 1–7, 2018.
- [19] Haoyu Li, Mingyu Wu, Binyu Zang, and Haibo Chen. Scissorgc: scalable and efficient compaction for java full garbage collection. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 108–121, 2019.
- [20] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.
- [21] Matchy JM Ma, Cho-Li Wang, and Francis CM Lau. Jessica: Java-enabled single-system-image computing architecture. *Journal of Parallel and Distributed Computing*, 60(10):1194–1222, 2000.
- [22] Martin Maas, Krste Asanović, Tim Harris, and John Kubiawicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. *Acm SIGPLAN Notices*, 51(4):457–471, 2016.
- [23] Martin Maas, Tim Harris, Krste Asanović, and John Kubiawicz. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.
- [24] Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu. Gerenuk: thin computation over big native data using speculative program transformation. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 538–553, 2019.
- [25] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 291–305, 2015.
- [26] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. Skyway: Connecting managed heaps in distributed big data systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24–28, 2018*, pages 56–69. ACM, 2018.
- [27] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 349–365, 2016.
- [28] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. *ACM SIGARCH Computer Architecture News*, 43(1):675–690, 2015.
- [29] OpenJDK. JEP 248: Make g1 the default garbage collector, 2017.
- [30] OpenJDK. JEP 310: Application class data sharing, 2018.
- [31] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.
- [32] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus prime: Accelerating data transformation in servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1203–1216, 2020.
- [33] Daniel J Scales and Monica S Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, pages 9–es, 1994.
- [34] Ioannis Schoinas, Babak Falsafi, Alvin R Lebeck, Steven K Reinhardt, James R Larus, and David A Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 297–306, 1994.
- [35] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337, 2017.
- [36] Xuanhua Shi, Zhixiang Ke, Yongluan Zhou, Hai Jin, Lu Lu, Xiong Zhang, Ligang He, Zhenyu Hu, and Fei Wang. Deca: a garbage collection optimizer for in-memory data processing. *ACM Transactions on Computer Systems (TOCS)*, 36(1):1–47, 2019.
- [37] Eishay Smith. Jvm-serializers, 2020.
- [38] Esoteric Software. Kryo, 2021.
- [39] Konstantin Taranov, Rodrigo Bruno, Gustavo Alonso, and Torsten Hoefler. Naos: Serialization-free RDMA networking in java. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 1–14. USENIX Association, July 2021.
- [40] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating application objects efficiently for heterogeneous computing. *ACM SIGARCH Computer Architecture News*, 44(3):53–65, 2016.
- [41] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 261–280, 2020.
- [42] Michal Wegiel and Chandra Krintz. Xmem: type-safe, transparent, shared memory for cross-runtime communication and coordination. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 327–338, 2008.
- [43] Weimin Yu and Alan Cox. Java/dsm: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, 1997.
- [44] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud’10, Boston, MA, USA, June 22, 2010*, page 95. USENIX Association, 2010.
- [45] Matthew J Zekauskas, Wayne A Sawdon, and Brian N Bershad. Software write detection for a distributed shared memory. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, pages 8–es, 1994.
- [46] Sizhuo Zhang, Hari Angepat, and Derek Chiou. Hgum: Messaging framework for hardware accelerators. In *2017 International Conference on ReConfigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2017.
- [47] Wenzhang Zhu, Cho-Li Wang, and Francis CM Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In *Proceedings. IEEE International Conference on Cluster Computing*, pages 381–388. IEEE, 2002.



Sift: Using Refinement-guided Automation to Verify Complex Distributed Systems

Haojun Ma Hammad Ahmad Aman Goel Eli Goldweber
Jean-Baptiste Jeannin Manos Kapritsos Baris Kasikci
University of Michigan
{mahaojun, hammada, amangoel, edgoldwe, jeannin, manosk, barisk}@umich.edu

Abstract

Distributed systems are hard to design and implement correctly. Recent work has tried to use formal verification techniques to provide rigorous correctness guarantees. These works present a hard choice, though. One must either opt for the power of refinement-based approaches like IronFleet and Verdi, at the cost of large amounts of manual effort; or choose the more automated approach of I4, IC3PO, SWISS and DistAI which give up the ability to prove refinement and the power and scalability that come with it.

We propose an alternative approach, Sift, that combines the power of refinement with the ability to automate proofs. Sift is a two-tier methodology that uses a new technique, *refinement-guided automation*, to leverage automation in a refinement proof and a divide-and-conquer technique to split a system into more refinement layers when necessary. This combination advances the frontier of what systems can be proven correct using a high degree of automation. Contrary to what was possible before, our evaluation shows that our novel approach allows us to prove the correctness of a number of systems with little manual effort, and to extend our proofs to include not just the protocols, but also an executable distributed implementation of these systems.

1 Introduction

Recently, formal verification has emerged as a potential alternative to the traditional approach of testing. The promise of formal verification—to eliminate all bugs by construction—is particularly attractive for distributed systems, which are notoriously hard to design and implement correctly.

Despite recent efforts, however, formal verification of distributed systems is still not ready for real-world applications. The most powerful techniques, such as IronFleet [34] and Verdi [63], rely on *refinement* proofs [1, 25, 42] to reason about complex systems and verify real implementations. Alas, the power of those techniques comes at a high cost: performing these refinement proofs manually requires large amounts of manual effort.

In an attempt to reduce the manual verification effort, the Ivy tool [56] proposes to express distributed protocols using decidable—and thus simpler to verify—reasoning [57]. The Ivy tool achieves remarkable automation, but still requires significant human effort to complete the proof. More recent approaches, like I4 [50, 51], IC3PO [27], SWISS [33] and DistAI [66], leverage model checking and SMT solvers to automate the most challenging part of proving the correctness of distributed protocols: finding an *inductive invariant*. Alas, this automation comes at the expense of expressiveness and applicability, because tools like I4 and DistAI were designed to prove properties of *monolithic* protocols which consist of a single layer. As such, they cannot prove refinement.

Refinement [1, 25, 42], however, is an essential concept in proving the correctness of real, complex systems. It allows us to prove the correctness of a system by showing that it is equivalent to a simpler, more abstract version of that system. The power of refinement comes in many forms:

Concise specification As Lamport has argued [45] and as IronFleet demonstrated, specifications should be written as simple, abstract state machines. Consider the specification of a Paxos-based State Machine Replication in IronFleet, where the goal is to prove that the entire service is linearizable. Expressing linearizability as a set of properties on the requests and responses is daunting and will likely yield a complex specification. Using refinement, the task is simple: just show that the entire service is equivalent to a single machine executing requests one at a time. Similarly, the sharded key-value store in IronFleet was simply proven equivalent to an abstract, logically centralized key-value store; i.e., a map.

Scaling to complex systems As IronFleet and Verdi demonstrated, the key to dealing with the complexity of a real system is to take a modular approach: split the proof into multiple layers and show that each layer refines the one above it. This is especially true when verifying actual implementations, as these tend to be much more complex than abstract protocols. In the absence of refinement, we are left with the task of reasoning about a single, monolithic system, whose complexity now becomes a limiting factor for both

manual and automated approaches.

Dealing with undecidability Even when one only cares about proving the correctness of the protocol, and not of the implementation, being unable to split a monolithic system into multiple layers can be a showstopper for automation. As Padon et al. demonstrated [55], some protocols may be undecidable by construction and thus not amenable to the automation of I4 and IC3PO. In these cases, one can use refinement to split the protocol into two layers, each of which is separately decidable [62].

We aim to get the best of what are currently two distinct worlds: the *power of refinement* (i.e. IronFleet-style proofs) but with only *a fraction of the manual effort* (i.e. using the automation of monolithic provers like I4, IC3PO, SWISS and DistAI). This combination allows us to not only achieve simple, concise specifications, but also to scale our proofs to more complicated distributed protocols, and even to distributed implementations.

To achieve this goal, we introduce Sift, a two-tier methodology that combines automated verification with a small amount of manual effort to push the boundary on the kinds of systems that can benefit from proof automation. Just like IronFleet before it, Sift is a *methodology*, not a tool. Its contribution is a way of structuring refinement proofs in order to leverage the automation of existing tools. Similar to how IronFleet guided developers to manually construct proofs based on the existing tools (TLA+ and Dafny), so does Sift show developers how to construct proofs that leverage the automation of more recent tools, like IC3PO and Ivy.

The first tier of Sift introduces a new technique, called *refinement-guided automation*, which leverages the automation of monolithic provers in the context of a refinement proof. At the high level, this technique enables the automation of refinement proofs between two layers by *encapsulating* the state of the upper, more abstract, layer into the state of the lower, more concrete layer. This encapsulation allows us to transform a two-layer refinement proof into a single-layer, monolithic proof that provers like I4, IC3PO, SWISS and DistAI can perform.

Leveraging automation to prove refinement is not always enough, though. Monolithic provers have their limits and thus some refinement proofs are just too complex to prove automatically. When that happens, we provide developers with an escape hatch. The second tier of the Sift methodology describes a divide-and-conquer technique for introducing intermediate layers, thus splitting a complex proof into chunks that are small enough for the prover to handle.

The Sift methodology applies refinement-guided automation within each refinement step and uses our divide-and-conquer technique to split a refinement step into smaller, more manageable steps. As a result, Sift allows us to apply, for the first time, automation to refinement-based proofs and scale to much harder problems than was previously possible. We use Sift to automate the verification of four distributed imple-

mentations, whose proof required minimal manual effort (less than five minutes, in most cases).

We further use our divide-and-conquer technique to prove the correctness of an implementation of Raft [54] and an implementation of MultiPaxos [43, 44] — a feat that was only possible before by providing a fully manual proof of correctness. Using Sift, we were able to automate most of the proof for both Raft and MultiPaxos. The manual effort required to complete the proof with Sift is not only significantly less than that of previous approaches, it is also much less reliant on having expertise in formal verification.

Overall, this paper makes the following contributions:

- We introduce *refinement-guided automation*, a technique that leverages the automation of monolithic-oriented tools to perform more complex, refinement-based proofs.
- We present a divide-and-conquer technique for splitting a complex refinement proof into smaller pieces, such that each piece is amenable to automated verification.
- We introduce Sift, a methodology that incorporates refinement-guided automation and our divide-and-conquer technique. We evaluate Sift on six distributed implementations and find that it allows us to prove their correctness in a mostly automated manner which drastically reduces the manual effort required compared to previous refinement-based approaches.

The rest of the paper is structured as follows. Section 2 discusses the tradeoff between automation and refinement. Section 3 recaps some background material, while Section 4 gives an overview of Sift. Section 5 introduces refinement-guided automation and Section 6 shows how to introduce intermediate refinement layers when needed. Section 7 evaluates the effectiveness of using Sift to automate the verification of a number of distributed implementations. Section 8 presents the limitations of Sift and discusses future work. Section 9 discusses related work and Section 10 concludes.

2 The Price of Automation

As discussed earlier, there are currently two approaches for verifying the correctness of distributed systems. The first is the powerful but manual approach of IronFleet and Verdi [34, 35, 63], where the developer uses refinement to show that a complex implementation is equivalent—through a series of layers or transformations—to an abstract specification.

The second approach is that of I4 [50], IC3PO [27], SWISS [33] and DistAI [66] which leverage the power of model-checking and SMT solving [5] to automatically prove the correctness of abstract system descriptions at the protocol level. These approaches aim to prove that a given safety property holds for the protocol at hand, by automatically identifying an *inductive invariant* that implies this safety property.

While such automation is undoubtedly a desirable property, it comes at a heavy price. In particular, I4, IC3PO, SWISS and DistAI can only perform *monolithic* proofs: they can prove

that a protocol—defined as a single layer—satisfies a given safety property. As we described in Section 1, this not only limits the type of specifications we can use, but also severely limits the scalability of the approach.

Most importantly, the scalability limitation is not an artifact of the implementation of *monolithic provers*—like I4, IC3PO, SWISS and DistAI—but rather inherent in their design. By asking the underlying solver to find an inductive invariant that supports the desired safety property, they essentially adopt an *all-or-nothing* approach: either the solver is powerful enough to find an inductive invariant or it is not. If we consider more and more complex systems, we soon reach a point where the solver is simply not powerful enough to find an inductive invariant.

In fact, a similar dichotomy presents itself when the protocol description has elements outside the *decidable* fragment of logic [47,55]. In several of these cases, the solver struggles considerably, even when it is trivial for a human to split the problem into decidable sub-problems. Without the ability to split this monolithic proof into multiple pieces, there is no middle ground. For example, I4 simply fails when the problem lies outside the decidable fragment, even though it is still possible to use refinement to split the protocol into two layers, each of which is separately decidable [62].

In this paper, we show that there exists a middle ground between the fully manual approaches that support refinement, like IronFleet and Verdi; and the automated-but-monolithic approaches, like I4, IC3PO, SWISS and DistAI. This middle ground, enabled by our novel Sift methodology, allows for refinement-based reasoning—and thus allows us to prove the correctness of complex distributed implementations—while making heavy use of automation to drastically reduce the amount of manual effort required compared to IronFleet and Verdi.

3 Background

3.1 Multi-Layer Refinement

Sift is heavily based on the notion of refinement. We will therefore first recap the notion of refinement and how it can be used to prove the correctness of complex systems.

A system P *refines* another system Q if the observable outputs produced by any execution of Q can also be produced by some execution of P . In the case of distributed systems, the only outputs that are visible to external observers are the messages produced by these systems.

In the simplest application of refinement, the developer writes two layers: a specification and an implementation. The specification is written as a simple, logically centralized state machine. In the case of a sharded key-value store, for example, the specification is a simple map, where the only possible actions are to put something to the map, or to get something from the map [34,35]. The developer then shows that the

implementation refines the specification, thus proving the correctness of the implementation.

In more complex systems, directly proving refinement from the implementation to the specification can be difficult [34,35,63]. In that case, the developer must insert one or more increasingly complex layers between the implementation and specification, thus creating a multi-layer structure, where each layer must be proven to refine the one above it. We explain how to design and insert intermediate layers in Section 6.

3.2 Automated Reasoning and Monolithic Provers

Traditional verification languages [4,46] rely on the developer to write a full proof, including a large number of manual annotations. As a result, approaches like IronFleet [34,35] and Verdi [63] incur a high proof-to-code ratio. To reduce this manual effort, Ivy [56] uses decidable logic to guarantee completeness. With Ivy, the developer only needs to find an *inductive invariant*—an invariant which is closed (inductive) under the system transitions—and the prover can automatically identify if this inductive invariant is correct. Ivy significantly simplifies the effort of proving the correctness of distributed systems, but finding such inductive invariants is still a non-trivial task that relies on human intuition and an intimate understanding of the system at hand.

To push the automation a step further, I4 [50] leverages the regularity of distributed protocols, so that the inductive invariant can be automatically inferred from a small, finite instance. Unfortunately, such a strategy only applies to monolithic protocols, not refinement proofs. Thus, I4 doesn't scale well when the system has a large state space and complex transitions. More recent tools [27,33,66] have followed the direction of using finite instances to guide the verification of distributed protocols. All these tools, however, apply only to monolithic proofs and cannot support refinement. We call such tools *monolithic provers*.

3.3 IC3PO: Our Monolithic Prover of Choice

The design of Sift does not rely on the internals of the monolithic prover that it uses. The refinement-guided automation technique of Sift can leverage any tool designed for automating monolithic, single-layer proofs. In fact, we previously tried I4 as the monolithic prover in Sift, but later found that IC3PO performs better. Our experience so far shows that IC3PO also outperforms SWISS and DistAI. As new and more powerful monolithic provers become available, Sift can adopt them to perform even larger refinement steps to further reduce manual effort. The next paragraph gives a short overview of IC3PO.

IC3PO [27,28] is a recently-developed prover that uses the synergistic relationship between symmetry and quantification to prove the safety of distributed protocols fully automatically,

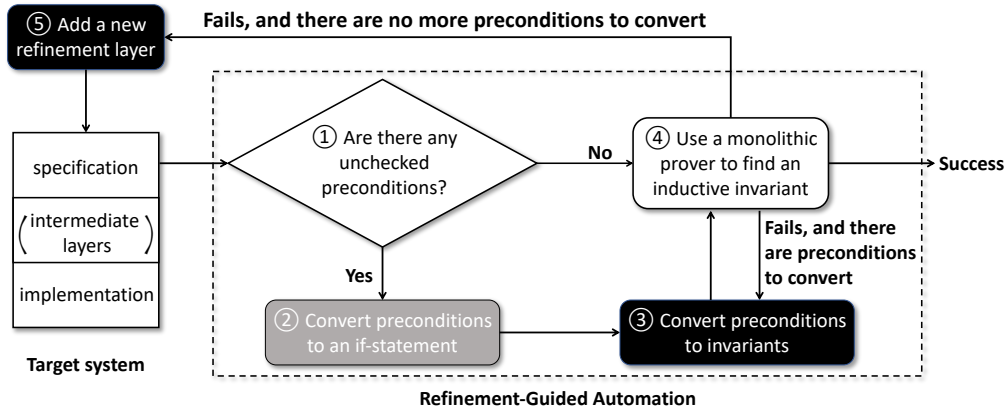


Figure 1: Summary of the Sift methodology. White boxes are fully automated, gray boxes indicate a trivial syntax change, and black boxes denote manual effort.

by inferring compact inductive invariants with both *universal* and *existential* quantifiers. At its core, IC3PO exploits the inherent regularity present in distributed protocols to significantly scale up IC3/PDR-style verification [10, 23] over finite instances of the protocol. Starting with an initial instance size, IC3PO systematically computes quantified inductive invariants over protocol instances of increasing sizes, until protocol behaviors *saturate*, concluding with an inductive proof that works for all instances of the protocol.

4 Overview of Sift

This paper introduces Sift, a methodology that allows reasoning about complex systems while still using a large degree of automation in proofs. Sift accomplishes this by employing a small amount of manual effort, when needed, to split the system into a number of layers, where each layer can be shown to refine the layer above it.

Figure 1 shows an overview of the Sift methodology. Initially, the developer starts with an implementation of the system, along with a specification, both written in the Ivy language [56]. If one were to use the Ivy prover, they would have to provide a manual proof of refinement between the specification and implementation. Sift, instead, introduces our *encapsulation* technique to merge the two layers into a single proof that our monolithic prover can attempt to solve.

Indeed, the first step of the Sift methodology is to attempt to prove refinement directly between the implementation and specification layers. If this proof is too much for the prover to handle, the developer adds an additional layer of refinement and tries again. Each additional layer of refinement splits the proof into smaller pieces that are more amenable to automation; but, of course, this comes at the cost of some manual effort, as the developer must manually introduce the new layer.

In the next two sections, we describe the Sift methodology in more detail. Section 5 describes how we can use the

Algorithm 1 Specification of the Sharded Hash Table (SHT)

```

1  function requests(R : request) : bool
2  function replies(R : reply) : bool
3  function map(K : key) : value
4  initialization {
5     $\forall R. requests(R) \leftarrow false$ 
6     $\forall R. replies(R) \leftarrow false$ 
7     $\forall K. map(K) \leftarrow 0$ 
8  }
9  action commit(req : request, rep : reply) = {
10   require rep.type = req.type
11   require rep.src = req.src
12   require rep.key = req.key
13   require req.type = read  $\Rightarrow$  rep.data = map(req.key)
14   if  $\neg requests(req)$  {  $\triangleright$  require  $\neg requests(req)$ 
15     if req.type = write {
16       map(req.key)  $\leftarrow$  req.data
17     };
18     requests(req)  $\leftarrow$  true;
19     replies(rep)  $\leftarrow$  true;
20   }
21 }
```

automation of a monolithic prover to perform a refinement proof between two layers (steps ①–④ in Figure 1). Section 6 presents the methodology for adding intermediate layers to the refinement structure (step ⑤).

Case Study: Sharded Hash Table Throughout this paper, we use the example of a Sharded Hash Table application (SHT) [34] to illustrate the Sift methodology. SHT implements a distributed key-value store, and consists of two layers, a specification layer and an implementation layer. As shown in Algorithm 1, the specification layer describes a key-value store as a simple *map* from keys to values. It maintains two local sets (modeled as boolean-valued functions, lines 1 and 2) to keep track of which messages (requests and replies) have

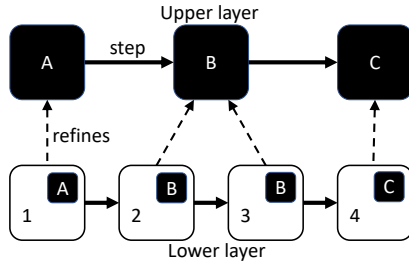


Figure 2: Encapsulation: to enable automatic refinement proofs, the state of the upper layer (A, B, C) is encapsulated inside the state of the lower layer (1, 2, 3, 4) refining it.

been sent. Initially, all keys are mapped to 0, and no messages have been sent. Requests can either be read requests or write requests. The only transition allowed by this specification is to *commit* a request *req* and its reply *rep*: i.e., perform the update (if this is a write request) and mark the corresponding messages as sent by setting *requests(req)* and *replies(rep)* to *true*. The specification layer consists of 32 lines of Ivy code.

In the implementation layer, every node contains a local hash table containing some subset of the total keys in the system and a delegation map. The node uses the delegation map to maintain its knowledge of where keys are stored on remote nodes. Each node can service a request using *get* and *set* actions for the keys that are locally stored, or use the delegation map to look up and forward requests to the appropriate node in the system if the requested key is not local. Nodes at the implementation layer can dynamically exchange sets of keys they are responsible for, by exchanging *delegate* messages (each carrying a key-value pair) among themselves. The implementation layer consists of 127 lines of Ivy code.

We aim to show that the implementation layer *refines* the specification, i.e., that any observable output produced by any execution of the implementation layer can also be produced by some execution of the specification. A key property is that for every key owned by a node at the implementation layer, the data matches the value stored in the specification. Additionally, every key is either owned by exactly one node in the system, or part of an in-flight *delegate* message.

5 Refinement-Guided Automation

We first explain the key high-level idea behind automating refinement proofs (step ④ in Fig. 1, Section 5.1). We then present what modifications Sift makes to the layers of a target system description to ensure that the correspondences between the layers are correctly represented before a proof is attempted (steps ①–③ in Fig. 1, Section 5.2).

Algorithm 2 Example of encapsulation in SHT

```

1  action set(req : request) = {
2    require req.type = write
3    owner ← delegation.get_owner(req.key)
4    if owner = me {
5      hash(req.k) ← req.v
6      rep ← create_reply(req)
7      call spec.commit(req, rep)
8      call network.send_reply(rep)
9    } else {
10   call network.forward_request(req, owner)
11   }
12 }
```

5.1 From Monolithic Proofs to Refinement

A key feature of Sift is that it uses the automation of monolithic provers to perform more complex, refinement-based proofs. As we explained in Section 2, monolithic (i.e. single-layer) proofs do not scale to complex systems, either due to complexity or undecidability. Yet, monolithic proofs are the only type of proof supported by these provers. The first innovation of Sift is that it converts a refinement proof between two layers into a monolithic proof which can be given as input to any monolithic prover.

We perform this transformation using our *encapsulation* technique, depicted in Figure 2. The idea of encapsulation is simple: if we want to show that a lower layer *L* refines an upper layer *U*, then we augment the state of *L* with the state of *U*. Additionally, whenever the state machine *L* makes a transition, the *encapsulated U* state also makes an upper-layer transition. In practice, this is expressed as a function call in Ivy, where the lower layer invokes a transition on its encapsulated state. For example, in the SHT application, the lower layer includes an encapsulated *spec* object (see Algorithm 1) and a lower-layer transition calls *spec.commit()* if it refines the *commit* transition of the upper layer.

Encapsulating the upper-layer state into the lower layer effectively creates a single, *augmented lower layer* that can be used to reason about the relation between the upper and lower layer. Most importantly, we can now leverage traditional single-layer provers to show whether a certain property—the refinement property—holds for this augmented lower layer.

Case study: refinement proof for the SHT Algorithm 2 shows a simple example of encapsulation at the implementation layer of SHT. To perform this encapsulation, the implementation layer imports (in Ivy) the specification layer. In this example of handling a *set* request, the program checks if this node (*me*) is the owner of the key in the request (line 4). If it is the owner, the implementation layer internally makes a call (line 7) to *spec.commit* (shown in Algorithm 1). This transition corresponds to the transition from state 1 to state 2 in Figure 2: the implementation layer transitions from state 1

to state 2, while each of these states encapsulates the corresponding upper layer state, indicating a transition from state A to state B at the upper layer.

If this node is not the owner, it simply redirects the request to the owner. Such an implementation layer transition does not entail a specification layer transition and so the code does not call *spec.commit* or any specification-level function. This is usually called a “stuttering” step of the specification layer—essentially a *no-op*—and corresponds to the transition from state 2 to state 3 in Figure 2.

To prove that the implementation refines the specification, we ask our monolithic prover to prove a simple property:

$$\forall R: reply, N: node. net.replied(R, N) \implies spec.replies(R)$$

This property says that any reply *R* sent to any node *N* at the network (implementation level) can only be present if the same reply *R* is present at the specification level. Since replies are the only observable outputs of the system, it ensures that every output of the implementation is also an output of the specification, thus ensuring that the implementation is indeed a refinement of the specification. Note that the reply message at the implementation layer is part of the network and thus modeled as *net.replied*(*M*, *N*).

5.2 Enforcing pre- and postconditions across layers

When calling functions from a lower layer to an upper layer, an upper-layer transition’s precondition must be met. The preconditions of the callee (in the upper layer) become postconditions (assertions) for the caller (in the lower layer) to check. For example, on line 13 of SHT’s specification (Algorithm 1), before committing a request, the precondition concerning the request, *req*, and the corresponding reply, *rep*, must be met:

$$req.type = read \implies rep.data = map(req.key)$$

This precondition ensures that every time a read request is committed, the data contained in the response must correspond to the data in the abstract map. Since it is the caller’s responsibility to guarantee that this precondition is met before committing the request, this precondition is effectively an assertion that needs to be checked by the monolithic prover. Unfortunately, the current state-of-the-art monolithic provers do not support checking these kinds of assertions, and can only find an inductive invariant for a safety property.

If we attempt to ignore this assertion check and let the monolithic prover prove the refinement property as is, the result could be unsound—i.e., the proof may go through even if the implementation is buggy. For example, let us consider again the refinement property for SHT:

$$\forall R: reply, D: node. net.replied(R, D) \implies spec.replies(R)$$

Without precondition checks, a buggy implementation can send a bogus reply message and call *commit* at the encapsulated specification layer. This would make the refinement property trivially inductive—since the *commit* call adds the message to the *replies*—without guaranteeing that contents of that message are correct.

To avoid this problem, Sift needs to consider the assertions in function calls to maintain soundness in automated refinement proofs. In the rest of this section, we explain how we transform the assertions to either conditionals (if/else) or invariants that the monolithic prover can reason about.

5.2.1 Converting Assertions to Conditionals

A straightforward approach to model assertions in function calls is to convert the callee to an *always-enabled action* using a conditional if/else block [35]. The developer can manually rewrite an assertion *P* as follows: if *P* holds, take the transition; otherwise, do nothing. In this context, the entire if/else block is always-enabled, in that it has no preconditions and can always be taken.

For example, the original SHT specification had a precondition $\neg requests(req)$ in the specification of the *commit* action, which we convert to an if-statement (Algorithm 1, line 14). This precondition ensures that the specification can never execute the same request twice.

The benefit of this approach is that it does not rely on any understanding of the system, which makes it very easy to implement. It has, however, two downsides. First, adding an if/else block in place of a precondition makes the proof a little harder for monolithic provers, since it is harder to find an inductive invariant for a weaker problem. Second, if the if-statement refers to ghost state—i.e., proof-related state that is not compiled to an executable—such as the sets corresponding to network messages, these if-statements are not compiled directly to executable code. Therefore, if there are any assertions that refer to ghost states at the implementation layer, we cannot rely on the approach of converting assertions to conditionals. In these cases, we need to convert them to invariants, as we describe below.

5.2.2 Converting Assertions to Invariants

A second, more involved approach to the problem is to convert these assertions into invariants. Doing so requires human intuition but reduces the difficulty for the monolithic prover. For every assertion that needs to be checked, there must be an invariant to support its proof. The key idea is simple: a programmer can trace backward through a function call from the upper layer (the callee) to the lower layer (the caller) to find the enabling precondition. For the SHT precondition example above, we observe that only the node who owns the key can commit the reply. Leveraging this observation, we can construct an invariant that if node *N* thinks it is the owner

of key K , the local value for key K at node N , which forms the reply, must match the value in the spec:

$$\forall N : \text{node}, K : \text{key}. \text{server}(N).\text{delmap}(K, N) \implies \text{server}(N).\text{hash}(K) = \text{spec.map}(K)$$

where $\text{server}(N).\text{delmap}(K, N)$ indicates that from the perspective of server N , the owner of key K is N (delmap stands for the delegation map). By maintaining this invariant, Sift can ensure the associated assertion will never be violated during the execution of the system. Note that the invariant is not necessarily inductive, but Sift leverages the automation of the monolithic prover to complete the proof.

Case study: converting assertions for the SHT The manual effort involved in the SHT proof requires converting seven assertions into two if-statements and five invariants. The assertion $\neg \text{requests}(r1)$ is converted from an assertion to an if-statement, as described in Section 5.2.1. On the other hand, the first three assertions (lines 10 to 12 in Algorithm 1) are already enforced by the implementation layer and do not need to be converted. We could further use the methodology described above in Section 5.2.2 to convert the fourth assertion (line 13) to an invariant, but it turns out that monolithic provers are powerful enough to complete the proof even if we simply convert it to a if-statement.

6 Introducing Intermediate Layers

We have discussed how to use automation to prove refinement between two layers. However, sometimes, the automation provided by the monolithic prover is not powerful enough to prove the desired refinement. This can happen either due to the complexity of the proof, or the presence of undecidable reasoning. When faced with such complex proofs, monolithic provers will either time out or run out of memory.

To perform such complex proofs, the solution is to introduce an intermediate layer (step ⑤ in Figure 1), thereby splitting the proof into two simpler refinement proofs: one refinement proof from the original lower layer to the intermediate layer, and another refinement proof from the intermediate layer to the original upper layer. By repeatedly using this proof-splitting technique until every refinement proof is automated, we effectively execute a *divide-and-conquer strategy* that allows us to tackle complicated refinements.

This idea is similar to IronFleet’s methodology of introducing an intermediate protocol layer to simplify the proof. In IronFleet, however, the developer needed to both *write* an intermediate layer and *manually prove* it correct. By contrast, Sift uses the automation of monolithic provers to dispense with most of the latter manual effort of writing the proof, and only requires the user to write intermediate layers—a much smaller effort than coming up with manual proofs.

Thankfully for developers, introducing an additional layer is done incrementally. The new layer is essentially a variation of the layer above or below it: either a more detailed version of the layer above it or a more abstract version of the layer below it. This helps keep the manual effort needed to introduce such layers small.

In the rest of this section, we discuss the strategies that we have developed and used to introduce intermediate layers, and walk through the process on a MultiPaxos example.

6.1 Intermediate Layers for Complexity

In most cases, the biggest challenge for a monolithic prover to automatically prove a refinement is its complexity. If the system is too complex, the prover either times out or runs out of memory. When this happens, we can split the refinement proof into two simpler refinement proofs by introducing an intermediate layer. We list here a number of ways in which such a split can simplify the proof burden. This list is extracted from our experience adding intermediate layers to facilitate refinement, and is not meant to be a complete enumeration of all possible layer-splitting strategies.

Abstract Away Messages Not Needed for Safety. Some of the messages used in the implementation may only be needed for liveness or performance, but not for safety. When trying to prove safety, those messages can be abstracted away in an intermediate layer: they are removed from the intermediate layer but kept in the implementation layer—itsself proven to be a refinement of the intermediate layer.

For example, in MultiPaxos the current leader needs to periodically broadcast a heartbeat message to indicate that it is still alive. This message is not needed for safety and can therefore be removed in an intermediate layer—though it is preserved in the implementation layer. The resulting intermediate layer is now simpler and thus easier to prove equivalent to the specification.

Merge Multiple Transitions into One Abstract Transition. Sometimes, the intermediate layer can take an abstract transition which is broken into multiple transitions in the low-level implementation.

For example, in MultiPaxos the learner can only receive one vote (two_b message) from an acceptor at a time. But what the learner really needs is a quorum of messages to learn a value. In this case we can merge multiple transitions of receiving each message separately into one abstract transition of receiving a quorum, and remove local variables for temporary results. This significantly simplifies the intermediate layer, with fewer state variables and simpler transitions.

Simplify Local State and Requirements for Transitions. Implementation layers have to take into account implementation constraints: for example, a node can only read its local

state when taking a transition; and it cannot access messages sitting in the network. But intermediate layers are essentially proof constructs and thus do not need to respect such implementation constraints.

For example, in MultiPaxos, a node needs to maintain an explicit local history of previous *two_b* votes to construct its *one_b* promise to a new leader, since a promise message depends on previous votes. In an intermediate layer however, a node can directly access all sent messages in the network, thus eliminating the need for this local history. Moreover, in an implementation a node can only read its local history, thus requiring a proof that the local history is consistent with sent messages. In the intermediate layer, since the node has access to all sent messages, it can directly check that the *one_b* promise is consistent with the vote messages, thus eliminating the need for this proof.

6.2 Intermediate Layers for Decidability

When the verifier returns an explicit decidability error, it means our refinement is not in the EPR decidable logic [47] and may take forever to check. Such an issue is typically resolved by introducing an intermediate layer and a ghost state (also known as a derived relation [55]) to hide the existential quantifier creating the undecidability [55, 62]. We apply a similar technique in Sift.

For example, in MultiPaxos an acceptor needs to send its last votes for different slots in a *one_b* message to a new leader to decide what value to propose. When a proposer becomes a leader, it needs to have a quorum of *one_b* messages, resulting in the following $\forall Round \exists Votes$ alternation:

$$\begin{aligned} \forall N : Node, R : Round. \text{quorum_of}(R).contains(N) \\ \implies \exists V : votes. \text{one_b}(N, R, V) \end{aligned}$$

The alternation of the \forall and \exists quantifiers, along with the inductive invariant, means that this proposition is outside the decidable logic of EPR. We leverage results from a followup work on Ivy [62], and introduce an intermediate layer to abstract away the payload (previous votes), thereby breaking the quantifier alternation. In this case, we only need an intermediate-layer state *joined_round*(*N*, *R*) to represent $\exists V. \text{one_b}(N, R, V)$.

7 Evaluation

We evaluate Sift by using it to formally verify the correctness of six implementations of distributed systems: a *leader election* protocol (Section 7.1), a *distributed lock* protocol (Section 7.2), a *two-phase commit* protocol (Section 7.3), a *sharded hash table* (SHT, Section 7.4), and two consensus protocols: *Raft* (Section 7.5) and *MultiPaxos* (Section 7.6). We use Ivy to implement these systems, and extract the executable

code to C++ using Ivy’s built-in translator. For the more complex systems (*SHT*, *Raft* and *MultiPaxos*), we also perform a performance evaluation (Section 7.7) to demonstrate our automated approach does not impact the performance of implementations.

For all systems in our evaluation, we consider crash failures and an asynchronous network, which can arbitrarily delay, drop, or duplicate messages. Both of these can be easily implemented in Ivy. Note that since Sift (like all its predecessors that also target automation) does not support liveness proofs, it does not need to explicitly reason about crash failures—a crash results in a machine no longer taking any steps and thus has no effect on safety properties.

We find that we are able to prove these complex systems with little manual effort within a reasonable memory and time budget, using IC3PO [27] as our monolithic prover. Our verification results are in Table 1. The complexity of different systems is illustrated by the number of different types that are needed to express state transitions for a given system. For example, for the leader election protocol, there are just two types: *node* and *id*. In contrast, MultiPaxos contains 14 different types, e.g., *round*, *inst*, *value*, *time*, *node*, etc.

We now give details about the proofs of the aforementioned systems, followed with a performance evaluation (Section 7.7) of three of the more complex resulting implementations (i.e., SHT, Raft and Paxos). We ran our performance experiments on a cluster where nodes have a 16-core Intel Xeon E5-2667 v4 @3.20 GHz processor and are connected with a 10 GB Ethernet connection running Ubuntu 16.04. All our implementation and artifact can be found in GitHub [49]

7.1 Leader Election

The *leader election* protocol aims to elect a unique leader from a ring with an unbounded number of nodes with unique integer IDs [13, 50, 56]. The specification layer dictates a single action the system can take: elect a node as the leader, under the condition that no other node is already the leader. This layer contains 13 lines of Ivy code.

In the implementation layer, the nodes are totally ordered in a ring so that every node has a next node. A node *n* has two valid actions: (a) periodically send its ID *idn*(*n*) to the next node in the ring; or (b) forward an ID *i* received from its predecessor if *i* > *idn*(*n*). Once *n* receives its *idn*(*n*), it knows that no other node in the system has a larger ID, and can now safely become the leader. The implementation layer consists of 28 lines of Ivy code.

To prove refinement between the implementation and the specification layers, we ensure that when a message stating that a leader is elected is sent in the implementation, the destination of the message should correspond to the leader node in the specification.

We perform a manual, albeit trivial, syntactic change to the specification layer to convert one precondition into an

System	Proof Effort	Refinement	# of types	Solution to Preconditions	# of Clauses in Invariant	Time (sec)	Memory (MB)
Leader Election	< 5 min	spec to impl	2	1 if-statement	6	196	1744
Distributed Lock	< 5 min	spec to impl	2	1 if-statement	8	111	425
Two-Phase Commit	< 5 min	spec to impl	4	3 if-statements	12	613	815
SHT	< 30 min	spec to impl	7	5 invariants, 2 if-statements	13	1021	856
Raft	1 person-month	spec to layer 0	6	manual			
		layer 0 to layer 1	6	15 invariants	22	787	4178
		layer 1 to impl	10	15 invariants, 1 if-statement	17	1239	2981
MultiPaxos	Previously proved	spec to layer 0	9	manual			
	3 person-weeks	layer 0 to layer 1	9	7 invariants, 2 if-statements	12	49	249
		layer 1 to layer 2	11	8 invariants, 8 if-statements	21	258	719
		layer 2 to layer 3	11	19 invariants	28	841	1935
		layer 3 to impl	14	19 invariants	25	196	398

Table 1: Summary of our six distributed systems; “spec” stands for specification, “impl” stands for implementation, and “layer i ” represents intermediate layers. The number of different types that are needed to express the state transition illustrates the complexity of different system.

if-statement, which takes less than 5 minutes. We then simply use Sift’s encapsulation technique to convert the refinement between the implementation and specification layers into a monolithic proof that is proven automatically by IC3PO.

7.2 Distributed Lock

The *distributed lock* protocol [34, 50, 56] models an unbounded number of nodes that transfer the ownership of a single lock. In this system, the ownership of a lock is associated with an ever-increasing epoch: only one node can own the lock at each epoch. This makes for a concise specification layer—12 lines of Ivy code—that only contains a lock history to indicate which node holds the lock at every epoch.

In the implementation layer, there are two possible transitions for a node: (a) transfer the lock if it holds the lock; or (b) accept the lock and jump to a higher epoch by sending a *locked* message to indicate ownership. This implementation has 35 lines of Ivy code.

The refinement property in this system is that all *locked* messages should have a corresponding node in the specification layer’s lock history.

The only manual effort involved in this proof is converting one precondition to an if-statement in the specification layer, which takes less than 5 minutes. After this transformation, we can use the encapsulation technique from Sift to convert the refinement between the implementation and specification layers into a monolithic proof, and prove the *locked* message is equivalent to the lock history.

7.3 Two-Phase Commit

The *two-phase commit* protocol [31] is used by a group of nodes, known as resource managers (RMs), to coordinate the decision on whether to abort or commit a transaction. The RMs vote to either commit or abort the proposed transaction and a transaction manager (TM) node is in charge of coordinating the decision-making procedure.

The specification layer of this system uses the Transaction Commit protocol by Lamport [30, Sec. 2] translated from TLA+ [45] to Ivy. The safety property does not allow a node to commit if another node aborts. The specification contains 54 lines of Ivy code.

The implementation of this system is an Ivy translation inspired by the TLA+ specification of Two-Phase Commit [30, Sec. 3]. This layer introduces a special TM node, which coordinates all RMs. An RM can send a *Prepared* message to the TM when transiting into the *prepared* state, or unilaterally decide to abort. Upon receiving a *Prepared* message from every RM, the TM can decide to commit, broadcasting a *Commit* message to every RM node. The receipt of a *Commit* message from the TM allows an RM to decide to commit the transaction. This implementation of two-phase commit has 110 lines of code.

The refinement property between the implementation and specification ensures that all RMs commit or abort at the same time between the implementation and the specification.

After a trivial syntactic change converting preconditions to three if-statements in the specification layer, this refinement property is proven automatically.

7.4 Sharded Hash Table (SHT)

The Sharded Hash Table protocol was previously introduced as a running example in Section 4. Its specification is a simple key-value map processing read and write requests. We can automatically prove the refinement from the implementation to the specification, after converting preconditions to five invariants and two if-statements to guide IC3PO, as detailed in Section 5.2. Compared to IronKV (IronFleet’s implementation of SHT), we simplify the delegate messages by transferring one key at a time. Transferring intervals of keys would require a loop iterating over keys and a loop invariant [59, 65], which cannot be found automatically by IC3PO.

The network interface for SHT is more complex than that of other systems. In particular, SHT’s network interface requires that messages are not delivered twice, so that requests can only be committed once and only one node at a time can own a key. As this is not part of refinement, we leverage an existing proof [53] for these requirements.

7.5 Raft

Raft [54] implements a shared log among nodes, which can be used to implement a fault-tolerant distributed service. The log is maintained as a set of (index, value) pairs.

Raft is a *term*-based protocol. In each term, a node can be elected as the leader, append values to the log, and replicate its log to other nodes by sending an *append* message. For safety, each node maintains its own log and only votes for a leader whose log is not earlier than its own. When the leader receives reply messages for its *append* message from a majority of nodes, the leader can consider all previous log entries committed. This strategy ensures that all future leaders contain the committed log.

At the specification layer, Raft can commit a prefix to an index in the leader’s log and ensure that only one value is committed at each index. The refinement property from the implementation to the specification ensures that they have the same log.

7.5.1 Intermediate Layers and Proof Effort

Our Raft implementation is similar to the previous Ivy implementation of Raft [62] with 212 lines of code. Due to undecidability, we could not refine the implementation to the specification directly. Instead, we build a first intermediate layer—layer 0—to separate the quantifier alternation (as outlined in Section 6.2). We tried to prove the refinement from specification to layer 0 automatically, but the inductive invariant contains complex quantifier alternations, which IC3PO was unable to handle. As a result, we manually prove the refinement from specification to layer 0. The refinement from spec to layer 0 took two person-weeks (including understanding the protocol). Layer 0 contains 143 lines of code.

From layer 0, the implementation is still too complex to refine directly using IC3PO. We introduce another intermediate layer, layer 1, to help IC3PO automatically prove the refinement. To write layer 1, we follow the strategies presented in Section 6, specifically by merging actions into one abstract action. In the abstract action a node can receive a quorum of messages at once, rather than receiving each of them individually in separate transitions. Layer 1 changes 57 lines from layer 0. We spent another two person-weeks to identify this intermediate layer and debug our implementation.

Overall, we were able to complete the proof of Raft in one person month, which compares favorably to the three person months needed by the original proof [62] written in Ivy. This reduction was the result of using a much higher degree of automation, by splitting the proof into layers and leveraging the power of IC3PO to prove each refinement between consecutive layers.

7.6 MultiPaxos

MultiPaxos [43, 44] is a common consensus protocol that is widely used in industry (e.g., Chubby [11], Megastore [2], and Spanner [19]). However, MultiPaxos is notoriously complex and difficult to verify.

At the specification level, *MultiPaxos* maintains an array of values; some that have been decided (i.e., agreed upon and finalized) and some that are empty. The only possible transition in the specification is to add a new decided value to this array. Similar to Raft, our refinement ensures that the implementation maintains the same values as the array in the specification.

The implementation of MultiPaxos is very similar to that of Raft but uses different strategies to ensure safety. In Raft, the leader can only be a node with the most up-to-date logs, while MultiPaxos relies on the messages from other nodes to generate an up-to-date log for the new leader.

7.6.1 Intermediate Layers and Proof Effort

Our design of the MultiPaxos protocol is inspired by previous work on expressing Paxos and MultiPaxos in the EPR decidable logic [55, 62]. Our evaluation uses the MultiPaxos implementation from [62], removing certain re-transmissions that are unnecessary for safety to simplify the refinement.

Since proving refinement directly between the implementation layer and the specification layer would introduce undecidability (see Section 6.2), we initially introduce a single intermediate layer, layer 0, to circumvent this undecidability. Moreover, as the refinement from the specification to layer 0 contains complex quantifier alternations that are too hard for IC3PO to prove automatically, we borrow the existing manual proof from Ivy. Layer 0 contains 88 lines of Ivy code.

After addressing undecidability concerns through layer 0, we found that a direct refinement from the implementation to

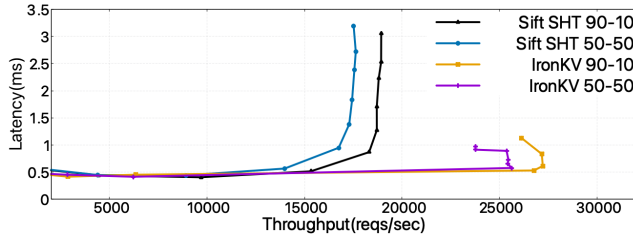


Figure 3: SHT performance

layer 0 remains infeasible for IC3PO. Using our divide-and-conquer technique, we added three additional intermediate layers to simplify this refinement. Following the strategies outlined in Section 6.1, we first added a layer 1 that abstracts away liveness messages and merges transitions to receive a quorum of messages. We augmented this by introducing a layer 2 that uses a local variable to track the current round, receives one *two_b* message, and keeps track of when a valid quorum can be formed. We then introduced a final intermediate layer that more closely resembles the implementation by using an array to track previous voted values for acceptors, and restricting a node to only receive one message during a transition.

With the addition of the four intermediate layers, Sift splits the complex refinement proof into manageable pieces, where each refinement between layers is amenable to automated verification. Producing the three intermediate layers (layers 1, 2, and 3) and converting the necessary preconditions to invariants is still a non-trivial task which takes about two person-weeks. About one third of the time is spent waiting for IC3PO to run out of time or memory, which indicates that another layer is needed (step ⑤ in Figure 1). While non-negligible, this manual effort is significantly less than the original attempt in Ivy, which was two person-months to refine layer 0 to the implementation [62].

7.7 Performance Evaluation

7.7.1 SHT Performance

We compare the throughput and latency of our verified Sift implementation of SHT with IronKV [34], as shown in Figure 3. IronKV is the closest verified implementation of a SHT that we could compare against. The SHT cluster was preloaded with 1,000 keys delegated evenly across the three nodes and serviced requests from an increasing number of clients in a closed loop. In one experiment, client processes send an even 50/50 mix of randomized GET and SET requests. We further increase the percentage of GET requests to 90%. IronKV scales about 25% better than our version of SHT. The disparity in performance between these two systems can be attributed to both unoptimized generated C++ from Ivy and design choices made in IronKV, which added extra manual proof complexity for the sake of performance purposes, such

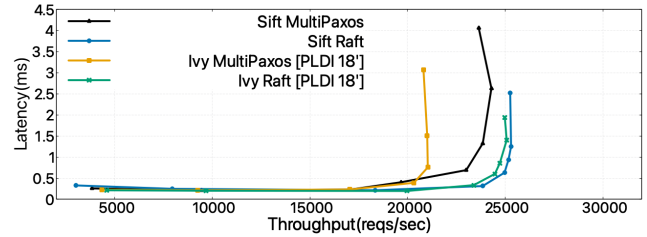


Figure 4: Raft and MultiPaxos performance

as an efficient delegation map data structure that each node maintains and consists of 833 lines of Dafny code.

7.7.2 Raft and MultiPaxos Performance

We evaluated the performance of the verified Sift implementations of Raft and MultiPaxos by varying the load of each system with an increasing number of clients submitting requests in a closed loop, as shown in Figure 4. For both systems, the experimental setup consists of three replicas on separate machines, with a fourth machine containing the client processes.

We tried to compare the performance of these systems with IronRSL, IronFleet’s verified Paxos-based replicated state machine library, but the performance results of IronRSL were not reproducible for a direct comparison. By re-running the original implementation from the IronFleet paper [34], we found the performance for IronRSL to be lower than originally reported [34, Sec. 7]¹. The performance of our implementation of MultiPaxos, which does not support batching, is comparable to the results reported for IronRSL in non-batch mode [34, Fig. 13].

We do compare both Raft and MultiPaxos with the Ivy-based manually-verified implementations [62]. The performance of Raft is almost identical to the version of Raft, but we find that our MultiPaxos system exceeds the performance of MultiPaxos from that work. These results show that the automation and reduced proof effort gained by using Sift does not impact the performance of either system.

8 Limitations and Future Directions

Our experience with Sift suggests that it advances what is possible in the realm of automated verification of complex systems. For all of its successes, however, there are still more steps to be taken in this direction.

- **Automating simple transformations.** While Sift greatly increases the automation of complex refinement proofs, parts of the methodology still require manual effort that could potentially be automated, such as converting assertions to if-

¹Even after close discussions with two of the IronFleet authors, this discrepancy was not resolved. They attributed this to possible code changes between what was originally evaluated and the currently available code.

statements and transforming to invariants through automatic computation of weakest preconditions [22].

- **Loop invariants.** Certain complex systems, such as SHT, may require loop invariants to prove optimizations that are added to enhance the performance of executable code. Loop invariants are similar to regular inductive invariants, in that both are inductive under some transitions. As described in Sections 7.4 and 7.6.1, any loop invariant in Sift must currently be written manually. In the future, we hope to add support for automatic derivation of loop invariants in Sift, by building further on the existing literature [59, 65].

- **Leveraging multiple monolithic provers.** As shown in recent works [27, 33, 66], different monolithic provers show complementary strengths in different scenarios. Since the design of Sift is independent of the choice of monolithic prover, we plan to employ a portfolio of monolithic provers in parallel to derive refinement proofs with even higher scalability.

9 Related Work

We now provide a summary on previous efforts relevant to applying formal methods to verify distributed systems.

Automated Verification. With the advancements in automated reasoning [6, 20, 36] and abstraction techniques [3, 16, 29], automatically verifying correctness through model checking [17, 58] has significantly improved in different domains, both for hardware [9, 10, 23, 26, 61] and software [3, 7, 8, 37, 41]. However, model checking still does not scale well to large complex systems, due to state-space explosion [18, 21].

More recently, several approaches [24, 27, 33, 38, 40, 50, 66] have extended induction-based model checking [10, 23] to automatically infer inductive invariants for infinite-state distributed protocols. I4 [50] leverages the regularity of distributed protocols, combining finite model checking with unbounded reasoning in distributed protocols. IC3PO [27], described in detail in Section 3.3, incorporates invariant generalization with model checking for better scalability. SWISS [33] derives an inductive invariant by performing an exhaustive search over candidate invariants in an optimized invariant search space. DistAI [66] uses a data-driven approach and is guaranteed to find a universally-quantified inductive invariant in finite time.

All the aforementioned techniques [24, 27, 33, 38, 40, 50, 66], however, target monolithic, single-layer verification, primarily at the protocol level, and cannot scale to detailed system implementations. In contrast, our approach combines these monolithic provers with the well-founded concepts of refinement [1, 25, 42] to scale verification all the way to complex executable implementations.

Systems Verification. Much effort has gone to verifying real systems, including OS kernels [15, 32, 39, 52], file, and storage systems [12, 14, 67]. These works provide strong guarantees of correctness, but at the cost of extensive manual effort; Sift,

by contrast, requires little manual proof effort while verifying systems of considerable complexity, such as MultiPaxos.

Within the realm of distributed systems, there have been attempts at manually verifying implementations of protocols [60, 64]. Ivy [56] requires the developer to iteratively refine an invariant until an inductive invariant is identified. IronFleet [34] and Verdi [63] have been used to verify practical implementations of distributed systems. In stark contrast to our work, all three approaches rely on considerable amounts of manual effort (in the order of person months) to complete a proof of correctness. Additionally, while IronFleet always uses three layers of refinement (i.e., specification, protocol, and implementation), most of the distributed systems we verify are refined directly from an implementation to a specification, with intermediate layers only added when needed to reduce the proof complexity for our monolithic provers.

More recently, Lorch et al. [48] presented Armada, a tool designed to verify concurrent programs. While Armada has some superficial similarities to Sift—namely the use of refinement and automation—it is in fact drastically different. It operates in an environment almost diametrically opposed to that of Sift: single-machine, multi-threaded code where communication happens via shared memory, as opposed to Sift’s sequential execution on a distributed system where communication happens via message passing. Additionally, while Armada makes heavy use of automation to generate proofs, it still requires its users to write significant parts of the proof—hundreds of lines of code—manually.

10 Conclusion

This paper introduces Sift, a novel two-tier methodology that combines the power of refinement with the ability to automate proofs. Sift decomposes the proofs of complex distributed implementations into a number of refinement steps, each of which is amenable to automation. We use Sift to prove the correctness of six distributed implementations—including the notorious MultiPaxos—none of which had an automated proof before. Our evaluation shows that this combination of refinement and automation lets us verify complex distributed implementations with little manual effort.

Acknowledgements

We thank the anonymous reviewers for their useful feedback in improving this paper. This work was supported by the National Science Foundation under grant No 2018915, and by an Amazon Research Award.

References

- [1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–

284, 1991.

- [2] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.
- [3] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *International Conference on Integrated Formal Methods*, pages 1–20. Springer, 2004.
- [4] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.
- [5] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
- [6] C. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.
- [7] D. Beyer. Software verification: 10th comparative evaluation (sv-comp 2021). *Tools and Algorithms for the Construction and Analysis of Systems*, 12652:401, 2021.
- [8] D. Beyer and M. E. Keremoglu. Cpatchecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*, pages 184–190. Springer, 2011.
- [9] A. Biere, N. Froylyks, and M. Preiner. Hardware model checking competition (HWMCC) 2020. <http://fmv.jku.at/hwmcc20>.
- [10] A. R. Bradley. Sat-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.
- [11] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, 2006.
- [12] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1054–1068, New York, NY, USA, 2019. Association for Computing Machinery.
- [13] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, 1979.
- [14] H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, page 270–286, New York, NY, USA, 2017. Association for Computing Machinery.
- [15] H. Chen, X. N. Wu, Z. Shao, J. Lockerman, and R. Gu. Toward compositional verification of interruptible os kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16*, page 431–447, New York, NY, USA, 2016. Association for Computing Machinery.
- [16] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.
- [17] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.
- [18] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [19] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google’s globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, Oct. 2012. USENIX Association.
- [20] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08*, pages 337–340, Berlin, Heidelberg, 2008. Springer.
- [21] S. Demri, F. Laroussinie, and P. Schnoebelen. A parametric analysis of the state-explosion problem in model checking. *Journal of Computer and System Sciences*, 72(4):547–575, 2006.

- [22] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
- [23] N. Een, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 125–134. FMCAD Inc, 2011.
- [24] Y. M. Y. Feldman, J. R. Wilcox, S. Shoham, and M. Sagiv. Inferring inductive invariants from phase structures. In *Computer Aided Verification*, pages 405–425, Cham, 2019. Springer International Publishing.
- [25] S. J. Garland and N. A. Lynch. Using i/o automata for developing distributed systems. *Foundations of component-based systems*, 13(285-312):5–2, 2000.
- [26] A. Goel and K. Sakallah. Model checking of verilog rtl using ic3 with syntax-guided abstraction. In *NASA Formal Methods Symposium*. Springer, 2019.
- [27] A. Goel and K. Sakallah. On symmetry and quantification: A new approach to verify distributed protocols. In *NASA Formal Methods Symposium*, pages 131–150. Springer, 2021.
- [28] A. Goel and K. A. Sakallah. IC3PO: IC3 for Proving Protocol Properties. <https://github.com/aman-goel/ic3po>.
- [29] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *International Conference on Computer Aided Verification*, pages 72–83. Springer, 1997.
- [30] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.
- [31] J. N. Gray. Notes on data base operating systems. In *Operating Systems*, pages 393–481. Springer, 1978.
- [32] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI’16*, page 653–669, USA, 2016. USENIX Association.
- [33] T. Hance, M. Heule, R. Martins, and B. Parno. Finding invariants of distributed systems: It’s a small (enough) world after all. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 115–131. USENIX Association, Apr. 2021.
- [34] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.
- [35] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: Proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, June 2017.
- [36] M. Heule, M. Järvisalo, M. Suda, T. Balyo, C. Sinz, and A. Biere. The international SAT Competitions web page. <http://www.satcompetition.org/>.
- [37] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009.
- [38] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham. Property-directed inference of universal invariants or proving their absence. *Journal of the ACM (JACM)*, 64(1):7, 2017.
- [39] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP ’09*, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.
- [40] J. R. Koenig, O. Padon, N. Immerman, and A. Aiken. First-order quantified separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*, page 703–717, New York, NY, USA, 2020. Association for Computing Machinery.
- [41] D. Kroening and M. Tautschnig. Cbmc-c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.
- [42] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.
- [43] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.
- [44] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001), pages 51–58, December 2001.
- [45] L. Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

- [46] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'10*, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.
- [47] H. R. Lewis. Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences*, 21(3):317–353, 1980.
- [48] J. R. Lorch, Y. Chen, M. Kapritsos, B. Parno, S. Qadeer, U. Sharma, J. R. Wilcox, and X. Zhao. Armada: Low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*, page 197–210, New York, NY, USA, 2020. Association for Computing Machinery.
- [49] H. Ma, H. Ahmad, A. Goel, E. Goldweber, J.-B. Jeannin, M. Kapritsos, and B. Kasikci. Sift Artifact. <https://github.com/GLaDOS-Michigan/Sift>.
- [50] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 370–384, 2019.
- [51] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah. Towards automatic inference of inductive invariants. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 30–36, 2019.
- [52] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying security invariants in expressos. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13*, page 293–304, New York, NY, USA, 2013. Association for Computing Machinery.
- [53] K. L. McMillan. non-duplicating ordered transport service. <https://github.com/microsoft/ivy/blob/master/doc/examples/sht/trans.md>.
- [54] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.
- [55] O. Padon, G. Losa, M. Sagiv, and S. Shoham. Paxos made EPR: decidable reasoning about distributed protocols. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–31, 2017.
- [56] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham. Ivy: safety verification by interactive generalization. *ACM SIGPLAN Notices*, 51(6):614–630, 2016.
- [57] R. Piskac, L. de Moura, and N. Bjørner. Deciding effectively propositional logic using dpll and substitution sets. *Journal of Automated Reasoning*, 44(4):401–424, Apr 2010.
- [58] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on programming*, pages 337–351. Springer, 1982.
- [59] G. Ryan, J. Wong, J. Yao, R. Gu, and S. Jana. Cln2inv: Learning loop invariants with continuous logic networks. In *International Conference on Learning Representations*, 2020.
- [60] N. Schiper, V. Rahli, R. Van Renesse, M. Bickford, and R. L. Constable. Developing correctly replicated databases using formal tools. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 395–406. IEEE, 2014.
- [61] B. L. Synthesis and V. Group. ABC: A system for sequential synthesis and verification. <http://www.eecs.berkeley.edu/~alanmi/abc/>, 2017.
- [62] M. Taube, G. Losa, K. L. McMillan, O. Padon, M. Sagiv, S. Shoham, J. R. Wilcox, and D. Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 662–677, 2018.
- [63] J. R. Wilcox, D. Woos, P. Panckekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. *ACM SIGPLAN Notices*, 50(6):357–368, 2015.
- [64] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2016*, page 154–165, New York, NY, USA, 2016. Association for Computing Machinery.
- [65] J. Yao, G. Ryan, J. Wong, S. Jana, and R. Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*, page 106–120, New York, NY, USA, 2020. Association for Computing Machinery.
- [66] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan. DistAI: Data-driven automated invariant learning for distributed protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 405–421. USENIX Association, July 2021.

- [67] M. Zou, H. Ding, D. Du, M. Fu, R. Gu, and H. Chen. Using concurrent relational logic with helpers for verifying the atomfs file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, page 259–274, New York, NY, USA, 2019. Association for Computing Machinery.

Faith: An Efficient Framework for Transformer Verification on GPUs

Boyuan Feng, Tianqi Tang, Yuke Wang, Zhaodong Chen, Zheng Wang, Shu Yang,
Yuan Xie, and Yufei Ding

University of California, Santa Barbara

{boyuan,tianqi_tang,yuke_wang,chenzd15thu,zheng_wang,shuyang1995,
yuanxie,yufeiding}@ucsb.edu

Abstract

Transformer verification draws increasing attention in machine learning research and industry. It formally verifies the robustness of transformers against adversarial attacks such as exchanging words in a sentence with synonyms. However, the performance of transformer verification is still not satisfactory due to bound-centric computation which is significantly different from standard neural networks. In this paper, we propose **Faith**¹, an efficient framework for transformer verification on GPUs. We first propose a semantic-aware computation graph transformation to identify semantic information such as bound computation in transformer verification. We exploit such semantic information to enable efficient kernel fusion at the computation graph level. Second, we propose a verification-specialized kernel crafter to efficiently map transformer verification to modern GPUs. This crafter exploits a set of GPU hardware supports to accelerate verification-specialized operations which are usually memory-intensive. Third, we propose an expert-guided autotuning to incorporate expert knowledge on GPU backends to facilitate large search space exploration. Extensive evaluations show that Faith achieves $2.1\times$ to $3.4\times$ ($2.6\times$ on average) speedup over state-of-the-art frameworks.

1 Introduction

Transformers [8, 21, 25, 32, 33, 38, 45] is an important category of neural networks (NNs) in machine learning research and industry. Transformers are first designed for natural language processing (NLP) and have achieved state-of-the-art accuracy across many NLP tasks such as neural machine translation [1, 26, 31] and sentiment analysis [7, 37, 48]. Due to its success, transformers have been widely used in many industrial products such as Facebook for hate speech detection [10] and Alexa for question answering [14]. Recently, transformers also show extraordinary accuracy for many computer vision tasks [9, 19, 44, 47, 55] and become the new trending model.

¹The project is open-sourced at <https://github.com/BoyuanFeng/Faith>

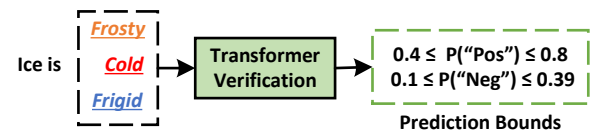


Figure 1: Illustration of transformer verification. Here, all perturbed inputs share the same prediction “positive” since the lower bound probability for “positive” (0.4) is higher than the upper bound probability for “negative” (0.39).

However, similar to prior NNs, transformers are also vulnerable to adversarial attacks that add imperceptible perturbations to input data for maliciously changing transformer predictions [2, 3, 16, 17, 22]. One specific example of adversarial attack is to exchange words (*e.g.*, cold) in a sentence with carefully selected synonyms (*e.g.*, frigid). This vulnerability may result in security concerns for real-world applications. For example, an intentionally crafted hate speech may spread widely on social network.

Transformer verification has been proposed to formally verify the robustness of a transformer against adversarial attacks [4, 18, 35, 42]. Given an input data x and a transformer $F(x)$, transformer verification identifies a maximal bound ϵ , such that all inputs x' that are “close” to the input data (*i.e.*, $|x' - x| \leq \epsilon$) cannot “mislead” the transformer (*i.e.*, $F(x) = F(x')$). A larger ϵ indicates better robustness. Early verification approaches [18] enumerate all possible inputs x' that satisfy $|x' - x| \leq \epsilon$ and conduct inference on each input to check predictions. These approaches show prohibitive latency due to the large number of inputs x' . Recent transformer verification [35, 42] avoids such enumeration by providing a single pair of lower and upper bounds for transformer predictions over all these inputs, as illustrated in Fig. 1. We can verify the robustness of a transformer if the lower bound of the correct prediction is higher than the upper bound of other predictions. The key computing pattern is a *bound-centric computation*, which computes a pair of inequality bounds for individual neurons. It first represents the input perturbations with inequality bounds over input neurons (*e.g.*, $x - \epsilon \leq x' \leq x + \epsilon$) and then propagates these bounds across layers to generate

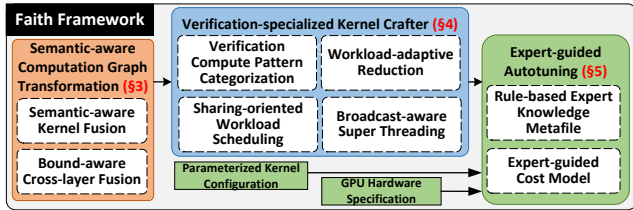


Figure 2: Overview of Faith Framework

the bounds for transformer predictions.

While transformer verification can formally verify the robustness of transformers, it also introduces high latency and limits its applications. In particular, transformer verification usually leads to second-level latency [35] in contrast to millisecond-level latency of standard transformers. We identify three challenges behind efficient transformer verification.

Lack of performance optimization over transformer verification computing patterns. Existing transformer verifications usually utilize the existing deep learning (DL) frameworks, such as PyTorch [30], which are designed for standard NNs. However, transformer verification shows significantly different computing patterns from standard NNs due to the nature of bound-centric computation. For example, when computing the upper bound of an output neuron, transformer verification needs to use the upper bound of the input neuron if the weight is positive; and the lower bound of the input neuron if negative. Straightforwardly deploying transformer verification to the existing DL frameworks usually leads to poor performance.

Lack of framework support for verifying diverse NN layers. Transformer verification shows large diversity in the bound computation for different types of NN layers such as projection layer with only perturbed features and self-attention layer with both perturbed weights and features. Even for the same type of NN layers, diverse upper bounds and lower bounds may be designed which requires different implementations. For example, Crown [52] utilizes two ReLU bound designs for generating more precise bounds for verification, where these bounds are selected dynamically according to the range of input neurons. This diversity makes it challenging to hand optimize GPU kernels in transformer verification.

Lack of verification-specialized adaptability towards modern GPUs. Transformer verification involves abundant memory-intensive operations such as reduction and broadcast. These memory-intensive operations can usually be significantly accelerated with rich architecture supports (*e.g.*, warp-level synchronized reduction) in modern GPUs. However, existing DL frameworks usually only focus on computation-intensive operations (*e.g.*, convolution) and ignore abundant optimization opportunities for memory-intensive operations. This leads to significant overhead in transformer verification with a large number of memory-intensive operations.

In this paper, we build **Faith**, the first framework for efficient transformer verification on GPUs. We show an overview

of the Faith framework in Fig. 2. First, we propose *semantic-aware computation graph transformation* to fully exploit fusion opportunities in transformer verification at the computation graph level. Our key insight is that transformer verification shows significantly different computing patterns (*e.g.*, two kernels for computing lower and upper bounds involve similar input data) from standard NNs. These computing patterns usually exhibit abundant data reuse opportunities. By exploiting such semantic information, Faith can fully harvest performance potential in transformer verification and achieve significant speedup over existing DL frameworks.

Second, we propose a *verification-specialized kernel crafter* to optimize transformer verification towards modern GPUs. Transformer verification contains abundant memory-intensive operations, such as elementwise computation, reduction, and broadcast. These operations may have complex dependencies and lead to performance bottlenecks. To this end, Faith automatically exploits a set of GPU architecture supports to improve the parallelism of such operations. Moreover, Faith introduces a set of optimizations to effectively mitigate memory access and improve performance by exploiting GPU memory hierarchies.

Third, we propose *expert-guided autotuning* to efficiently search optimized implementations in the large search space. Existing DL frameworks [6, 54] usually conduct autotuning in a hardware-agnostic approach where an ML-based cost model is deployed to implicitly learn hardware impact over performance from scratch. Instead, we propose a rule-based expert knowledge metafile to explicitly provide a small set of hardware characterizations and an expert-guided cost model to incorporate the expert knowledge. Faith exploits these two components to achieve efficient schedule exploration in the large design space of transformer verification.

In summary, this paper makes the following contributions:

- We build Faith, the first efficient framework to optimize the performance of transformer verification on GPUs.
- We propose a set of verification tailored system optimizations. In particular, we design a *semantic-aware computation graph transformation* to identify and exploit novel fusion opportunities for transformer verification, a *verifier-specialized kernel crafter* to effectively map transformer verification kernels to GPU backends, and an *expert-guided autotuning* to incorporate a set of expert knowledge on modern GPU architecture to guide large design space exploration.
- Extensive experiments show that Faith achieves up to $3.4\times$ speedup ($2.6\times$ on average) over state-of-the-art frameworks.

2 Related Work and Motivation

In this section, we first introduce the background of transformer verification (§2.1). Then, we discuss related work on DL frameworks (§2.2). Finally, we present opportunities

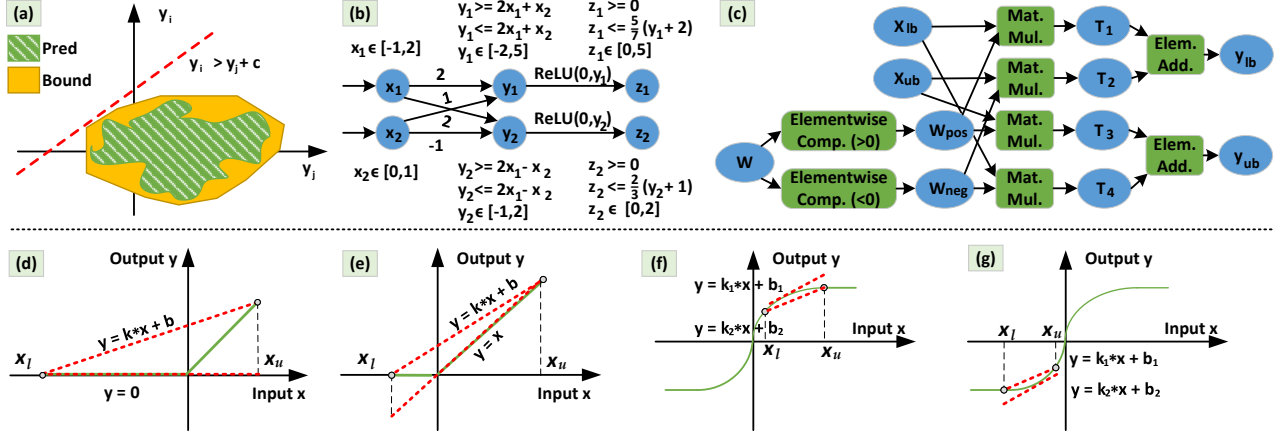


Figure 3: Illustration of transformer verification. (a) model prediction and verification bound; (b) an example of verifying a model with a fully connected layer and a ReLU layer; (c) computation graph of projection layer in transformer verification; (d)-(e) two types of bounds for ReLU layer; (f)-(g) two types of bounds for the *Tanh* layer.

and challenges for efficient transformer verification on GPUs ([§2.3](#)).

2.1 Transformer Verification

Standard Transformers. Transformer [8, 25, 38, 45] takes a sentence as input and predicts a label for this sentence (e.g., hate speech or benign speech). Given a sentence with *Length* tokens, we usually first map each token to a pretrained embedding [28] of dimension *Dim_in* and represent the feature of a sentence as a tensor of shape $Length \times Dim_in$. For a batch of sentences, we have input feature X as a tensor of shape $Batch_size \times Length \times Dim_in$, where *Batch_size* is the number of sentences in a batch. Since the number of tokens varies across sentences, *Length* is set to the maximal number of tokens over all sentences in a batch.

A transformer has three types of operators. The first type is the *elementwise operator* that applies computation on individual feature scalars. For example, on each scalar x in the input feature, we have $ReLU(x) = \max(0, x)$ and $Tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$. The second type is the *matrix multiplication operator* that takes an input tensor X , a weight matrix W , and generates an output tensor $Y = XW$. We note that these two types are similar to operators in prior neural networks. The third type is the *dot product operator*, which is the key idea behind the transformer model. Informally speaking, it takes two input tensors Q and K of the same shape $Batch_size \times Length \times Dim_in$. Then, it computes an output tensor $Y = Q^T K$ of shape $Batch_size \times Length \times Length$ to measure the pairwise similarity between individual words in a sentence. This similarity can significantly improve the learning capacity of the model and the prediction accuracy.

Adversarial Attack on Transformers. Adversarial attack [2, 3, 15, 16, 17, 22] identifies small perturbations to input data X that can change the transformer prediction. Formally, consider a transformer $f(\cdot)$, an input sentence X , and a tolerable input perturbation bound ϵ , where the transformer cor-

rectly classifies X as a label i (e.g., hate speech). In other words, the sentence has label i and $y_i > y_j$ for any $j \neq i$ where y_j is the predicted probability. Adversarial attack identifies a slightly perturbed sentence $X' = X + \eta$ such that $\eta \in B(0, \epsilon)$ and there exists a label j (e.g., benign speech) such that $y_i < y_j$. This perturbed sentence X' is an *adversarial example*.

Transformer Verification. Transformer verification [4, 18, 35, 42] computes a maximum bound ϵ and mathematically proves that there does not exist an adversarial example X' within the ϵ -ball of X (i.e., $(X' - X) \in B(0, \epsilon)$). Verifying transformers is challenging since transformers are essentially non-convex functions. The key idea of transformer verification is to utilize linear bounds as an approximation to NN predictions. We illustrate transformer verification at the model prediction layer in [Fig. 3\(a\)](#). Given these linear bounds, transformer verification can simply check if the predictions inside the bounds satisfy certain linear requirements, such as $y_i > y_j + c$, where c is a positive number. As illustrated in [Fig. 3\(a\)](#), this bound-based approach is sound since the linear bound covers the non-convex area of NN predictions.

We show an example of bound-centric computation of transformer verification in [Fig. 3\(b\)](#). Consider a fully connected layer $Y[j] = \sum_{i=1}^n W[j, i] \cdot X[i]$ where $Y[j]$, $W[j, i]$, and $X[i]$ are scalars. Here, we skip the index for batch size and length for notation simplicity. A formal summary of notations can be found in [Table 1](#). For each neuron $X[i]$, there is a lower and an upper bound

$$X[i] \geq X_{lb}[i] + X_{lw}[i] * \vec{\epsilon}, \quad X[i] \leq X_{ub}[i] + X_{uw}[i] * \vec{\epsilon}$$

where $X_{lb}[i]$ and $X_{ub}[i]$ are scalars, $X_{lw}[i]$, $X_{uw}[i]$, and $\vec{\epsilon}$ are vectors. For the input neurons, we have $X_{lb}[i] = X_{ub}[i] = X[i]$, $X_{lw}[i]$ and $X_{uw}[i]$ are one-hot vectors with 1 at the index i and 0 at other indices. Given this linear bound, we can compute *concretized* bounds for each neuron as

$$X_l[i] = X_{lb}[i] - \epsilon * \|X_{lw}[i]\|, \quad X_u[i] = X_{ub}[i] + \epsilon * \|X_{uw}[i]\| \quad (1)$$

where $\|\cdot\|$ computes the norm with reduction operations.

Table 1: Notations in transformer verification.

W	Transformer weights. Shape: $Dim_in \times Dim_out$
X	Input feature tensor. Shape: $Batch_size \times Length \times Dim_in$
X_{lb}, X_{ub}	The tensor of lower and upper bound bias of input features. Shape: $Batch_size \times Length \times Dim_in$
X_{lw}, X_{uw}	The tensor of lower and upper bound weights of input features. Shape: $Batch_size \times Length \times Dim_in \times Dim_out$
X_l, X_u	The tensor of concretized lower and upper bounds of input features. Shape: $Batch_size \times Length \times Dim_in$

When computing the bounds for output neuron $Y[j]$, we note that bound computation depends on the sign of weights $W[j, i]$. In particular, we have upper bounds $Y_{ub}[j]$ as

$$\begin{aligned}
 Y[j] &\leq Y_{ub}[j] + Y_{uw}[j] * \bar{\epsilon} \\
 &= \left(\sum_{W[j,i] \geq 0} W[j,i] \cdot X_{ub}[i] + \sum_{W[j,i] < 0} W[j,i] \cdot X_{lb}[i] \right) \\
 &\quad + \left(\sum_{W[j,i] \geq 0} W[j,i] \cdot X_{uw}[i] + \sum_{W[j,i] < 0} W[j,i] \cdot X_{lw}[i] \right) * \bar{\epsilon}
 \end{aligned} \tag{2}$$

The lower bounds can be computed in a similar way. This bound computation (Eq. 2) is significantly different from standard NN computation since it explicitly considers the sign of weights. Previous transformer verification directly exploits the standard DL frameworks to build a computation graph (Fig. 3(c)) for computing bounds, which leads to inefficient memory access and computation overhead. We will discuss the opportunities and challenges of efficient transformer verification in §2.3.

For the same NN layer, diverse bound computation designs may still be developed to provide tighter bounds on NN predictions. We illustrate two types of bounds for the ReLU layer in §2(d)-(e) and two types of bounds for the Tanh layer in §2(f)-(g). A tighter bound (*i.e.*, less space between linear bounds and ReLU function) is preferred to provide a better linear bound approximation to NN prediction. For example, consider the concretized lower bound $X_l[i]$ and upper bound $X_u[i]$ for an input neuron $X[i]$, when we have $abs(X_l[i]) > abs(X_u[i])$, linear bound in Fig. 3(d) is preferred over the linear bound in Fig. 3(e) since the former one provides a tighter approximation. This diversity in bound design adds more complexity to developing frameworks for transformer verification.

2.2 Deep Learning Frameworks on GPUs

GPUs have been widely exploited to accelerate deep learning workload [13, 39, 40, 46, 49]. Efficiently mapping deep learning workloads to the GPU computing and memory hierarchy is usually the key to improve performance [11, 23, 41, 50, 51]. GPU computing hierarchy contains threads, warps, and blocks [29]. Each block has multiple warps and each warp has exactly 32 threads that compute with single-instruction-multiple-data (SIMD). GPU memory can be generally treated as a hierarchy of registers, shared memory, and global memory. Accessing registers is much faster than accessing shared memory, which is faster than accessing global memory. Each thread can only

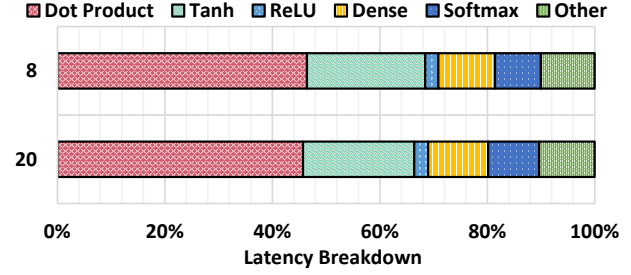


Figure 4: Latency breakdown of transformer verification on sentences with length 8 and 20. Here, we show the latency of verifying individual operators such as dot product and Tanh.

access its own registers and threads in a block cannot access shared memory from other blocks.

Many DL frameworks [6, 30, 54] have been developed recently to efficiently support NN workload on GPUs. Early works such as PyTorch [30] take user-specified computation graphs for neural networks and maps towards hand-tuned kernels on backend platforms (*e.g.*, GPUs). However, this approach usually builds upon kernels developed for standard NNs and cannot efficiently support transformer verification computation. Recent works, such as TVM [6] and Anso [54], can automatically generate such backend kernels based on a set of heuristic rules on fusion and operator optimizations. However, these heuristic rules are developed specifically for standard NNs. Naively incorporating these rules into transformer verification may lead to unsatisfactory performance due to the significant difference in computation patterns. For example, Fig. 3(c) shows the computation graph for utilizing the kernels of standard NNs on transformer verification. This approach leads to heavy sparsity and redundant memory access. In particular, only half of the elements in W_{pos} and W_{neg} are non-zero values, leading to 50% sparsity. To this end, we build Faith, the first framework for efficient transformer verification on GPUs.

2.3 Opportunities and Challenges

In this section, we introduce optimization opportunities and challenges in enabling efficient transformer verification.

We show the latency of verifying individual transformer operators in Fig. 4. We profile this latency breakdown based on the state-of-the-art transformer verification implemented with PyTorch [30]. We have three major observations. First, dot product accounts for around 45% latency. Dot product takes two input tensors Q and K where both inputs may be perturbed during adversarial attack, which is significantly different from matrix multiplication that only one input (*i.e.*, feature X) may be perturbed. This adds complexity to the verification of dot product operators [35] and longer latency. Second, elementwise operators such as Tanh and ReLU account for a large portion of latency in transformer verification. This is significantly different from standard NNs where elementwise operators can usually be fused with remaining operators and show low latency. Third, we observe that ma-

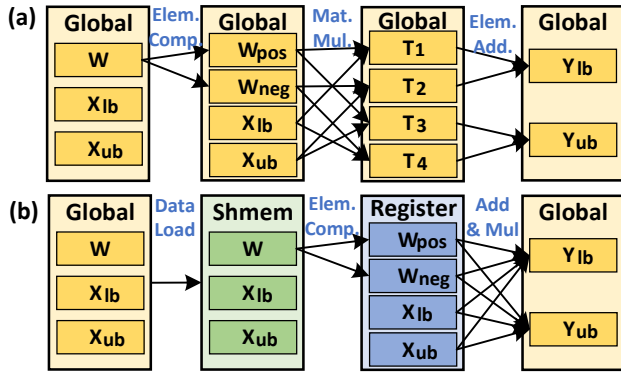


Figure 5: Illustration of Semantic-aware Kernel Fusion. We show the memory access pattern before and after applying semantic-aware kernel fusion in (a) and (b), respectively.

trix multiplication and softmax accounts for certain latency.

Opportunities: There are two major opportunities to accelerate transformer verification. The first opportunity is to exploit the semantics of transformer verification to minimize redundant memory access and computation. Our investigation shows that transformer verification has rich semantic information (e.g., 50% sparsity in W_{pos} and W_{neg}), which can be exploited to accelerate transformer verification. The second opportunity is to exploit the modern GPU architectures to efficiently support diverse computing patterns in transformer verification. One example is to accelerate abundant reduction computation in Eq. 1.

Challenges: Although these ideas sound promising, the efforts to realize the benefits are non-trivial due to several challenges. First, transformer verification shows significantly different computing patterns from standard NNs. Straightforwardly borrowing optimizations for standard NNs such as kernel fusion can hardly bring similar benefits. Second, while exploiting GPU architecture supports may bring benefits, we still need specialized designs as a synergy between architecture and specialized computing patterns. Moreover, exploiting advanced GPU architecture supports will add more complexity to the search space of optimized kernels which motivates novel autotuning optimizations.

3 Semantic-aware Computation Graph Transformation

In this section, we propose *semantic-aware computation graph transformation* for efficient transformer verification. We first propose **semantic-aware kernel fusion** to fuse kernels within a transformer layer. It contains two novel types of fusions – *weight-pairing based fusion* and *double bound based fusion*. Then, we propose **bound-aware cross-layer fusion** to efficiently fuse kernels across transformer layers.

3.1 Semantic-aware Kernel Fusion

The semantic-aware kernel fusion fuses operators in a single transformer layer to minimize memory access. Different from standard transformers, a single layer in transformer verification usually involves multiple kernels to compute the bounds adaptively to the sign of weights, as discussed in §2.1. Existing transformer verification [35, 42] usually uses a set of GPU kernels developed for standard transformers to serve the need for transformer verification. We illustrate the memory access pattern of this baseline approach in Fig. 5(a). These kernels need to independently read data from the global memory of GPUs and lead to heavy memory overhead. Moreover, these kernels fail to exploit semantic information in transformer verification and show heavy redundancy during memory access. For example, baseline approaches usually first split the weight matrix W into two weight matrices W_{pos} and W_{neg} according to weight signs and then use each matrix for computing lower and upper bounds. Here, these two split matrices W_{pos} and W_{neg} have the same shape of $M \times N$ as the weight matrix W . However, reading these matrices independently requires loading $2MN$ scalars, which leads to redundant memory access.

We propose semantic-aware kernel fusion to minimize such memory overhead by exploiting transformer verification semantics and GPU memory hierarchies (i.e., global memory, shared memory, and registers). We illustrate our semantic-aware kernel fusion in Fig. 5(b). Our key insight is to first load data collaboratively from global memory and only distinguish data semantics (e.g., W_{pos} and W_{neg}) at the register level to mitigate redundant memory access. In particular, we identify *weight-pairing based fusion* and *double bound based fusion* as the two most important semantics in transformer verification.

Weight-pairing based fusion. We first propose weight-pairing-based fusion to mitigate redundant memory access when reading W_{pos} and W_{neg} . Our key observation is that the zero values in W_{pos} are exactly the position of non-zero values in W_{neg} . Formally, we have $W_{pos} + W_{neg} = W$. To this end, instead of using an operator to split weight matrix W into W_{pos} and W_{neg} , we first load the matrix W from global memory to shared memory without distinguishing the sign of individual scalars. Then, we split the weight matrix W into W_{pos} and W_{neg} when loading data from shared memory to registers, as illustrated in Fig. 5(b). In our design, we only need to load MN scalars from global memory, which leads to significantly reduced memory access compared with loading $2MN$ scalars in baseline approaches.

Double bound based fusion. Our second optimization is a double-bound-based fusion. One important semantics in transformer verification is to multiply the same weight matrix with lower and upper input bounds (e.g., X_{lb} and X_{ub}) to compute the output bounds (e.g., Y_{lb} and Y_{ub} in Fig. 5(b)). Meanwhile, when computing the bound for output neurons, we usually need to read both lower and upper bounds for

computation. For example, when computing the upper bound of output neurons, we need to read upper bound when weight is positive and read lower bound when weight is negative. Suppose the input bounds X_{lb} and X_{ub} have shape $N \times K$, we need to load $4NK$ scalars during transformer verification.

Instead, we propose to fuse the computation of lower and upper bounds such that the lower and upper bounds only need to be loaded once to save memory access. In particular, we first use threads across GPU blocks to collaboratively load tiles of input matrices from global memory to shared memory, which can be accessed by different GPU threads. Here, we use shared memory to enable data sharing across GPU threads since different threads may multiply the same input bound scalar with different weight scalars (e.g., multiplying the first row in X_{lb} and X_{ub} with various columns in W). Then, each thread loads independent data from shared memory to registers and directly accumulates output bounds Y_{lb} and Y_{ub} in registers. We note that this design further improves performance by eliminating the redundant global memory access during generating Y_{lb} and Y_{ub} .

3.2 Bound-aware Cross-layer Kernel Fusion

Bound-aware cross-layer kernel fusion fuses the verification of kernels across multiple transformer layers to further minimize memory access. Existing frameworks for accelerating standard NNs usually rely on a set of rules to fuse kernels. One popular example is to fuse convolution kernel with the following elementwise kernels (e.g., ReLU kernel for elementwise comparison with 0). However, these rules usually cannot be applied to fuse kernels for transformer verification. For example, verifying the ReLU kernel requires first a concretization operation with a global reduction to compute the concretized bounds for a neuron and then applies different computation according to the concretized bounds (see §2.1).

To this end, we propose a set of rules for cross-layer kernel fusion in transformer verification. In particular, we recognize three types of operators. The first type is *input-reduction-compute* that conducts reduction or concretization operation on the input data before computation. One example is verifying nonlinear activation functions such as *ReLU* and *Tanh* that requires concretized bounds to apply different computation. Another example is the *softmax* operator that computes a global summation for normalization. The second type is *strict-elementwise* that contains only elementwise computation and does not require concretization or global summation. The third type is *dense-computation* such as matrix-matrix multiplication kernels. In our cross-layer kernel fusion design, we can always fuse a *dense* operator with its following *strict-elementwise* operator. However, we cannot fuse *dense* operator with *input-reduction-compute* due to the concretization or reduction operation. In addition, we can fuse *input-reduction-compute* with its following *strict-elementwise* operator. Finally, we can fuse multiple *strict-elementwise* operators (e.g.,

elementwise addition and multiplication).

4 Verification-specialized Kernel Crafter

In this section, we propose a verification-specialized kernel crafter to efficiently map transformer verification towards modern GPUs. We exploit intrinsic properties (e.g., abundant reduction operations) of transformer verification which are significantly different from standard transformer operators. One major challenge in building the kernel crafter is the large diversity in verification designs across operators (see Fig. 3(d)-(g)). To tackle this challenge, we first propose a *verification pattern categorization* to abstract such diversity and provide a small set of computing patterns over verification of diverse operators. Then, we propose three optimizations to efficiently support these computing patterns of transformer verification.

4.1 Verification Pattern Categorization

While there are diverse bound designs across different operators, we characterize transformer verification into four typical computing patterns. Based on this characterization, Faith can abstract the diversity in bound designs into a combination of computing patterns and exploit optimizations towards individual computing patterns for improving performance. Similar to standard NNs, one important computing pattern is *generalized matrix multiplication (GEMM)* when verifying projection layers and fully connected layers. Matrix multiplication is the major bottleneck in standard NNs and has been well-optimized by existing DL frameworks. Besides GEMM, transformer verification introduces three other time-consuming computing patterns, which are highlighted as follows:

The first computing pattern is *generalized vector reduction*. One typical source of generalized vector reduction is concretization that computes the norm and generates the concretized lower and upper bounds for individual neurons (see Eq. 1). Formally, consider a matrix $X = [\vec{x}_1, \vec{x}_2, \dots, \vec{x}_m] \in \mathbb{R}^{m \times n}$ where $\vec{x}_i = [x_{i,1}, x_{i,2}, \dots, x_{i,n}]$ are vectors of length n . The generalized vector reduction computes an output $Y = [y_1, y_2, \dots, y_n] \in \mathbb{R}^n$ that satisfies

$$y_i = \text{reduction}(\vec{x}_i) = \sum_{j=1}^n f(x_{i,j}), \quad i \in \{1, 2, \dots, m\} \quad (3)$$

Here, $f(x)$ is an elementwise function that takes a scalar input and generates a scalar output. One example for $f(x)$ is x^2 when computing the L_2 norm for input vectors.

The second computing pattern is *generalized elementwise multiplication* which appears frequently when verifying elementwise operators such as ReLU and Tanh. Formally, consider a concretized lower bound $l \in \mathbb{R}^{m \times n}$ and an upper bound $u \in \mathbb{R}^{m \times n}$ where $l_{i,j}$ and $u_{i,j}$ are concretized lower and upper bounds for the neuron at position (i, j) . Let $X \in \mathbb{R}^{m \times n}$ be the input values. The generalized elementwise multiplication

computes an output $Y \in \mathbb{R}^{m \times n}$ that satisfies

$$y_{i,j} = f(l_{i,j}, u_{i,j}) * x_{i,j}, \quad i \in \{1, 2, \dots, m\}, j \in \{1, 2, \dots, n\} \quad (4)$$

Here, transformer verification introduces a function $f(\cdot, \cdot)$ that takes the lower and upper bounds for an input neuron and computes a scaling parameter which is multiplied with the input value of this neuron. One example is the tangent line between the concretized lower and upper bounds when verifying Tanh layer, which accounts for more than 20% latency as we profiled in Fig. 4. Another example is $f(l_{i,j}, u_{i,j}) = 1$ when verifying ReLU layer and $l_{i,j}$ is non-negative. While $f(\cdot, \cdot)$ shows large diversity across operators, we stress that the same computing pattern is shared across these operators such that a uniform framework can be applied to improve performance.

The third computing pattern is *generalized scalar-vector multiplication*. This computing pattern exists widely when verifying dot products in the self-attention layer of transformers. This computing pattern accounts for more than 40% latency in transformer verification, as discussed in Fig. 4. Formally, consider a vector $S = [s_1, s_2, \dots, s_m] \in \mathbb{R}^m$ and a matrix $X = [\vec{x}_1, \vec{x}_2, \dots, \vec{x}_m] \in \mathbb{R}^{m \times n}$, where s_i are scalars and $\vec{x}_i = [x_{i,1}, x_{i,2}, \dots, x_{i,n}]$ are vectors of length n . The generalized scalar-vector multiplication computes an output $Y = [\vec{y}_1, \vec{y}_2, \dots, \vec{y}_n] \in \mathbb{R}^{n \times n}$ that satisfies

$$\vec{y}_i = f(s_i) * \vec{x}_i = [f(s_i) * x_{i,1}, f(s_i) * x_{i,2}, \dots, f(s_i) * x_{i,n}], \quad i \in \{1, 2, \dots, m\} \quad (5)$$

Here, $f(\cdot)$ is a function that takes a scalar input and generates a scalar output.

Generability to diverse NN operators. Faith can effectively support verifying diverse NN operators such as SiLU and Leaky ReLU. Our key insight is that verifying diverse NN operators usually share the same generalized computing pattern while the concrete computation formula might be different. For example, $SiLU(x) = \frac{x}{1+e^{-x}}$ is an activation function that has significantly different concrete computation formula from $ReLU(x) = \max(0, x)$. However, both verifying SiLU and ReLU can be treated as the generalized elementwise multiplication (Eq. 4) and the same optimizations can be applied to improve performance.

In the following sections, we first demonstrate a *workload-adaptive reduction* to improve the performance of generalized vector reduction (Eq. 3). We then propose a *sharing-oriented workload scheduling* to improve the performance of generalized elementwise multiplication (Eq. 4). Finally, we demonstrate *broadcast-aware super threading* to efficiently support the generalized scalar-vector multiplication (Eq. 5).

4.2 Workload-adaptive Reduction

Transformer verification contains abundant reduction operations where a sequence of scalars are summed up into one scalar. One common reduction operation is the concretization operation that computes the concretized lower and upper

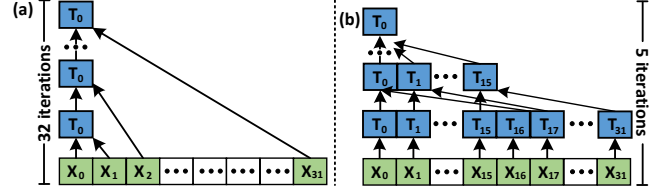


Figure 6: Illustration of Workload-adaptive Reduction. (a) Sequential Mode; (b) Parallel Mode. Here, x_i and T_i are the i -th data and thread, respectively.

bounds for individual neurons, as discussed in §2. Another common reduction operation is the softmax operation that is applied in each self-attention layer for measuring the relationship between individual words. These reduction operations pose challenges between parallelism and data locality. One baseline approach is to use a single thread to read and accumulate a sequence of scalars as illustrated in Fig. 6(a). However, this approach usually leads to low parallelism and fails to exploit abundant threads in GPUs. For example, we need 32 iterations to accumulate 32 scalars. Another baseline approach is to first split this sequence of scalars into multiple chunks and allocate one thread to each chunk for accumulation. Then, each thread writes the accumulated results for each chunk to global memory and uses an additional thread to finally accumulate the sum of each chunk. While this approach improves parallelism, it requires expensive global memory access and high overhead.

Workload-adaptive Reduction with length $n = 32$. We propose a *workload-adaptive reduction* to fully exploit GPU memory hierarchies and the inter-register communication functionalities. We illustrate our design in Fig. 6(b). Our design achieves high parallelism by enabling multiple threads for reduction simultaneously. Meanwhile, we avoid the expensive data communication through global memory and exploit only efficient registers. In particular, we use 32 threads (*i.e.*, a warp) to read these 32 scalars simultaneously from global memory. Considering these 32 scalars are consecutive in global memory, we can efficiently load them with 32 threads through coalesced memory access. Then, we exploit the specialized instruction `_shfl_down_sync` to directly communicate data in registers across individual threads. As illustrated in the parallel mode of Fig. 6(b), our design involves only five iterations of cross-thread data communication to generate the final accumulated result, rather than the 32 iterations in the sequential mode of Fig. 6(a).

Workload-adaptive Reduction with Arbitrary Length n . For an arbitrary length n , one naive approach is to repeatedly use 32 threads to reduce 32 scalars and then use 1 thread to accumulate the final results. However, this approach may lead to unnecessary communication across threads. Suppose we are accumulating a vector of length $n = 32k$, we need 5 iterations for reducing every 32 scalars, leading to $5k$ iterations in total for accumulating the vector. Instead, we propose

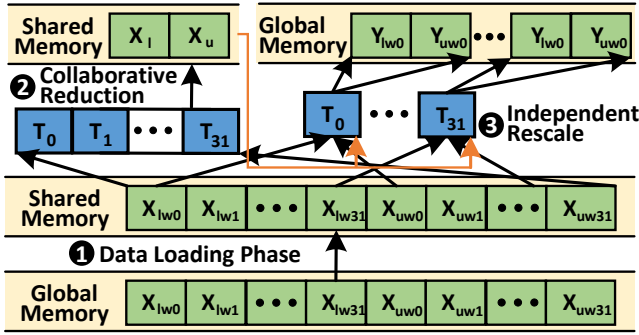


Figure 7: Illustration of sharing-oriented workload scheduling

a *hybrid mode* to minimize the number of iterations while still achieving high parallelism. In particular, we first split the input sequence into chunks where each chunk contains 32 scalars. Then, we use 32 threads to read one chunk simultaneously from global memory and accumulate individual chunks iteratively. For example, the 1-st thread accumulates the 1-st scalar in each chunk. Here, the accumulation is conducted in registers and does not require communication across threads. Finally, we apply a single 5-iteration reduction across 32 threads. In total, our design has only $k + 5$ iterations which are significantly less than $6k$ iterations in the naive approach.

4.3 Sharing-oriented Workload Scheduling

We propose *sharing-oriented workload scheduling* to efficiently verify elementwise operators. Different from standard transformers, verifying elementwise operators, especially non-linear ones (e.g., ReLU and Tanh), accounts for a large portion of latency in transformer verification as we discussed in Fig. 4. Verifying these operators usually first requires computing a concretized lower bound X_l and upper bound X_u for each input neuron and then computes the bounds for the output neuron. Different signs of concretized input bounds usually lead to different computations for output bounds, which could easily lead to warp divergence and unsatisfactory performance. Moreover, when computing the output bound weights (i.e., Y_{lw} and Y_{uw}) for a neuron, we need to repeatedly use the same input bounds which leads to extra memory overhead.

To efficiently verify elementwise operators, we propose *sharing oriented workload scheduling* to minimize memory access and improve performance. Our key observation is that the same set of input bound weights X_{lw} and X_{uw} are used to compute the concretized input bounds X_l and X_u , while these input weights are also used for computing the output bound weights Y_{lw} and Y_{uw} . Instead of repeatedly loading X_{lw} and X_{uw} , we can exploit the GPU memory hierarchies to cache X_{lw} and X_{uw} and minimize the global memory access to improve the overall performance.

As illustrated in Fig. 7, we use a set of $T (= 32)$ threads to first (Step 1) load input bound weights X_{lw} and X_{uw} from global memory to shared memory. Here, T is a hyper-

parameter to balance the parallelism and compute intensity, which will be selected in §5. Then (Step 2), these T threads load input bound weights from shared memory and collaboratively compute the concretized lower and upper bounds X_l and X_u , following our design in §4.2. These concretized lower and upper bounds are stored in shared memory which can be accessed by individual threads. Finally (Step 3), each thread independently loads individual X_{lw} and X_{uw} scalars from shared memory and rescales according to the concretized bounds X_l and X_u . Here, all threads in a warp are computing the output bound weights for the same neuron and the concretized input bounds are the same across threads in a warp. Thus, all threads in a warp can apply the same rescaling computation and avoid warp divergence. We also note that input bound weights are only loaded once from global memory which mitigates redundant global memory access.

4.4 Broadcast-aware Super Threading

We propose *broadcast-aware super threading* to efficiently support generalized scalar-vector multiplication, as discussed in Eq. 5. One naive approach is to use one thread to read a scalar s_i and a vector \vec{x}_i and computes the generalized scalar vector multiplication $f(s_i)\vec{x}_i$. However, this approach fails to exploit the parallelism opportunities in generalized scalar vector multiplication. Another approach is to split the vector \vec{x}_i into multiple chunks and use one thread for each chunk. However, this approach requires threads to repeatedly read the same scalar s_i from global memory and shows redundant memory access.

Instead, we propose a broadcast-aware super threading to achieve high parallelism while minimizing memory access. We consider two types of super threading for generalized scalar vector multiplication. The first type is a group of 32 threads (i.e., a warp for one vector). When using 32 threads to compute the multiplication between a scalar s_i and a vector \vec{x}_i , these 32 threads can read the scalar s_i once, broadcast across threads with modern GPU memory, and compute $f(s_i)$ simultaneously. Based on this broadcast, we can mitigate the redundant memory access that each thread repeatedly read the same scalar s_i . The second type is a group of $32t$ threads (i.e., t warps for one vector). In this case, we use one warp to read the scalar s_i and use shared memory to broadcast s_i across warps.

5 Expert-guided Autotuning Optimization

Considering the large design space of optimization towards GPUs, one natural question arises: *Can we effectively incorporate hardware knowledge to find optimal operator implementation?*

Existing works such as TVM [6] and Ansor [54] usually autotune operator implementations in a hardware-agnostic way. In particular, these works extract implementation-specific parameters such as tiling size and use a cost model to implicitly

learn the relationship between these parameters and performance. However, there are two drawbacks in this hardware-agnostic approach. First, there is a complex interaction between implementation and the hardware properties, which could be hard to be implicitly learned by the cost model. For example, existing works [12, 20, 24, 43] on hand-tuning large matrix-matrix multiplication operators usually maximize the number of registers in use to improve cache performance. However, this optimization is also limited by the number of registers for each GPU thread since exceeding such limitation may lead to register spilling [27] and a significant performance drop. A careful reasoning on the interaction between the implementation-specific parameters (e.g., the number of registers for caching data) and the hardware properties (e.g., the number of registers per thread) is usually necessary to maximize the performance. To tackle this challenge, we propose an *expert-guided autotuning optimization* to automatically reason both implementation-specific parameters and hardware properties. In particular, we have the following designs.

Rule-based Expert Knowledge Metafile. We propose a *rule-based expert knowledge metafile* to capture hardware properties. This metafile only needs to be set once for each type of GPUs and requires limited manual efforts. In particular, we consider two types of rules. The first type is *hard rules* which represents hardware limitation such as the maximal shared memory size and the maximal number of registers per thread. Violating these rules may lead to significant performance drop such as register spilling. The second type is *soft rules* which represents intrinsic trade-offs related to the hardware properties such as the number of streaming multiprocessors (SM) and the number of threads per SM. One typical design choice is the number of threads per block which will be mapped to threads on the same SM. Allocating more threads per block usually leads to better parallelism for the sub-task assigned to a block. However, allocating more threads per block may also hinder executing multiple blocks on the same GPU SM hardware and lead to worse overall parallelism.

Expert-guided Cost Model. We propose an *expert-guided cost model* to automatically tackle the complex interaction between implementation-specific parameters and hardware properties. Given a set of candidate operator implementations, we have two phases to select the optimal implementation. The first phase is to estimate the shared memory and register usage for each candidate. We rule out candidates that consume more shared memory and registers than hardware capacity, as specified in the expert knowledge metafile.

The second phase is to train a cost model for the remaining candidates and use the cost model to select the best candidate. We use XGBoost [5] as the cost model. It takes as input the implementation-specific parameters (e.g., tiling size) for candidates and the hardware properties (e.g., shared memory size). We use the cost model to predict the latency of candidates and select top-k candidates with low latency. Finally, we profile the latency of these top-k candidates on GPUs and use

Dataset	#Train	#Val	#Test	Length		
				min	mean	max
SST	67,349	872	1,821	4	25	62
YELP	560,000	0	38,000	5	98	128

Table 2: Dataset statistics

the profiled latency to further finetune our cost model. We repeat this procedure for a pre-defined iterations (=5 by default) and select the implementation with the lowest latency.

When training the cost model, we construct the training dataset as follows. We randomly select a small number of candidates and use their implementation-specific parameters and the hardware properties as feature X . Then, we profile the latency of each candidate implementation on GPUs as the label Y . We collect these (X, Y) as the training dataset to train the cost model.

6 Evaluation

In this section, we comprehensively evaluate Faith over various datasets and GPU backends. We first present our experiment setup in §6.1. Then, we show the overall speedup on end-to-end transformer verification in §6.2. Finally, we provide more optimization analysis on individual transformer layers in §6.3.

6.1 Experiment Setup

Baselines. We compare Faith with the state-of-the-art transformer verification [35] based on PyTorch. We further compare with TVM [6] and Anso [54], as stronger baselines. TVM and Anso are two state-of-the-art deep learning compilers for standard neural networks. We feed the pytorch model into TVM and Anso through relay frontend [34] which will automatically optimize transformer verification performance. While TVM and Anso take minutes to compile an operator implementation, we do not incorporate this compilation latency and record only inference latency for a fair comparison.

Datasets. We evaluate two popular datasets, Yelp [53] and SST [36], following the setting in state-of-the-art transformer verification [35]. These two datasets are widely used in the natural language processing for analyzing sentiment in languages. We summarize the statistics of these two datasets in Table 2. SST dataset contains 67,349 training sentences, 872 validation sentences, and 1,821 testing sentences. In SST dataset, there are 4 to 62 tokens in each sentence and the average number of tokens in a sentence is 25. YELP dataset contains 560,000 sentences as training data and 38,000 sentences as testing data. In YELP dataset, there are 5 to 128 tokens in each sentence and the average number of tokens in a sentence is 98.

Transformer Networks. We evaluate Faith on transformer networks with 1 to 6 layers to demonstrate the performance on large models. Following popular transformer settings, each transformer layer has 4 attention heads and an embedding size

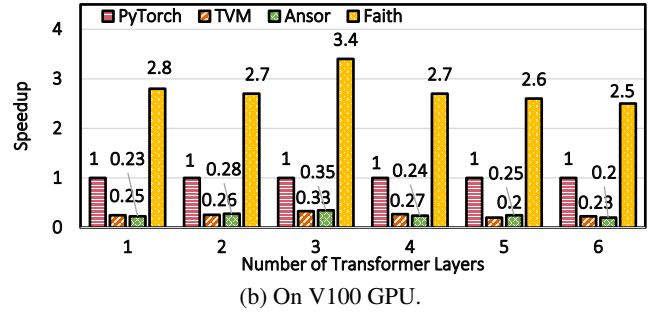
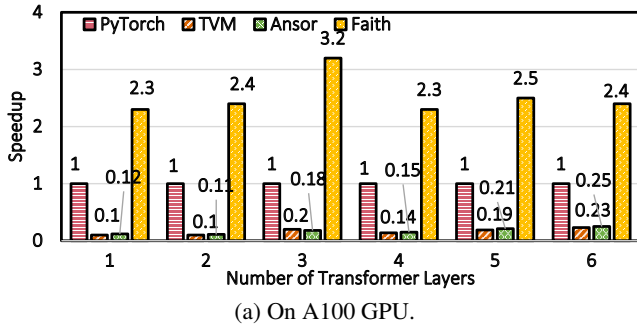


Figure 8: Overall speedup on SST dataset.

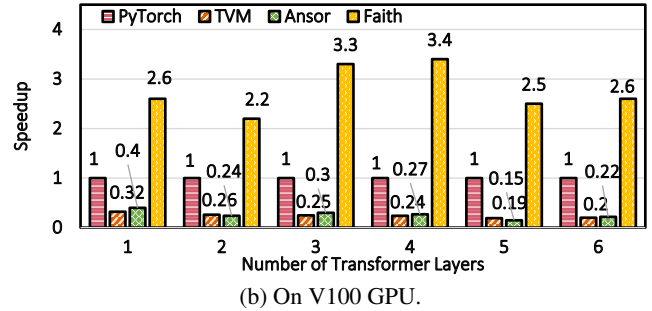
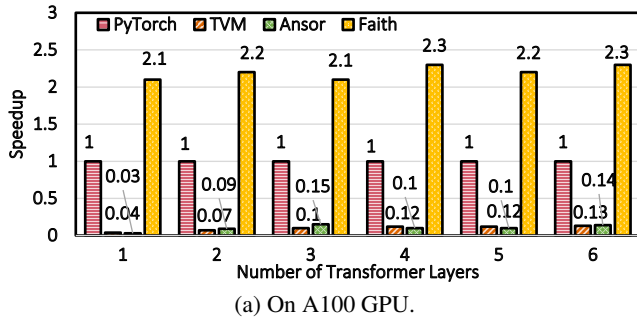


Figure 9: Overall speedup on Yelp dataset.

of 128. Furthermore, we study the Faith performance under diverse embedding sizes in §6.3.

Experiment Configuration. We evaluate with an NVIDIA A100 GPU and an NVIDIA V100 GPU to show Faith performance on various GPU backends. The host server with A100 GPUs is an AMD EPYC 7742 64-Core Processor and runs Ubuntu 20.04 with CUDA 11.3. The host server with V100 GPUs has 32 cores of Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz and runs Ubuntu 16.04 with CUDA 10.1.

6.2 Overall Performance

We show the overall speedup on SST dataset and Yelp dataset in Fig. 8 and Fig. 9, respectively. We show the performance improvement over transformers with diverse numbers of layers from 1 to 6, which covers popular settings in the natural language processing domain. While the length of input sentences may have an impact on the performance improvement, we show the averaged speedup over all testing sentences in this section and study the impact of sentence length in §6.3. We compare Faith with the PyTorch baseline following existing transformer verification open-source implementations [35]. We further compare Faith with two state-of-the-art deep learning frameworks (*i.e.*, TVM and AnsoR) to provide a comprehensive comparison, as we discussed in §6.1.

We show the overall speedup on SST dataset and A100 GPU in Fig. 8(a). Compared with PyTorch, we observe $2.3\times$ to $3.2\times$ speedup ($2.5\times$ on average). We contribute this performance improvement to our semantic-aware computation graph transformation (§3) and verification-specialized kernel crafter (§4). We further observe $17.2\times$ and $15.9\times$ speedup over TVM and AnsoR, respectively. The main reason is that

TVM and AnsoR focus on optimizing standard neural networks and fail to efficiently support verification-specific computing patterns, as discussed in §2.2. While Faith and these three baselines show different performance, we stress that the same verification bounds are generated, and the only difference resides in system optimizations. Comparing across different numbers of transformer layers from 1 to 6, the performance improvement remains similar around $2.5\times$. This result shows that Faith can efficiently support transformer verification with diverse numbers of transformer layers. We show the overall speedup on SST dataset and V100 GPU in Fig. 8(b). We have similar observation about the results on A100 GPU which shows that Faith can effectively adapt to diverse GPU backends, thanks to expert-guided autotuning optimization (§5).

We show overall speedup on Yelp dataset and A100 GPU in Fig. 9(a). Sentences in YELP dataset has 5 to 128 tokens (98 on average), which is longer than sentences in SST dataset with 4 to 62 tokens (25 on average). This provides an opportunity to show Faith performance on long sentences. Overall, we observe $2.1\times$ to $2.3\times$ speedup ($2.2\times$ on average) when comparing with the PyTorch baseline. We also observe $26.7\times$ and $28.3\times$ speedup on average over TVM and AnsoR, respectively. This speedup is similar to the performance improvement on SST dataset and shows the good generality of Faith over diverse input data. We also have similar observations on Yelp dataset and V100 GPU in Fig. 9(b).

6.3 Optimization Analysis

In this section, we show speedup from individual Faith optimizations. We first show speedup on *verification of matrix*

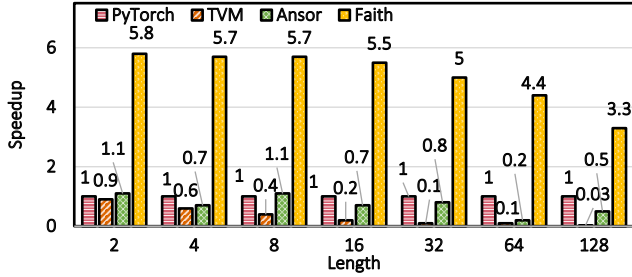


Figure 10: Speedup on verification of matrix multiplication over the diverse lengths. Embedding Size: 128.

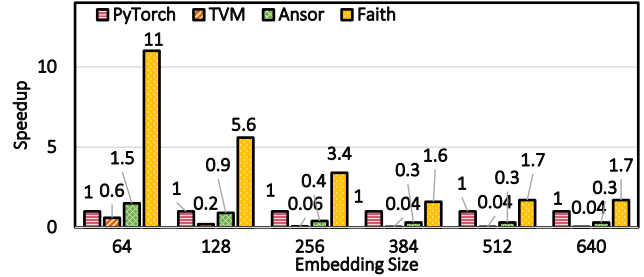


Figure 11: Speedup on verification of matrix multiplication over the diverse embedding sizes. Length: 16.

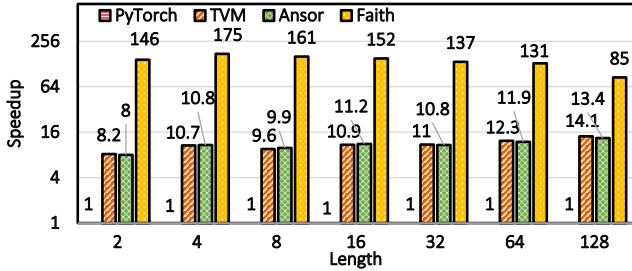


Figure 12: Speedup on verification of ReLU over the diverse lengths. Embedding Size: 128.

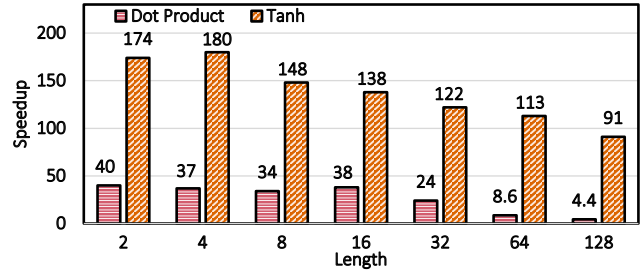


Figure 13: Speedup on verification of Tanh and dot product over the diverse lengths. Embedding Size: 128.

multiplication over the diverse lengths and diverse embedding sizes. Verification of matrix multiplication plays an important role in verifying projection layers and fully connected layers in transformers. Then, we show the benefits on verification of ReLU, verification of TVM, and verification of Tanh, which in total accounts for around 70% latency in transformer verification. Since we observe similar performance on A100 GPU and V100 GPU, we focus on A100 GPU and omit results on V100 GPU in this section due to page limits.

Performance benefits on verification of matrix multiplication. We show speedup on verification of matrix multiplication over the diverse lengths in Fig. 10. We study the speedup over diverse lengths from 2 to 128, following the setting in the popular natural language processing datasets as summarized in Table 2. Overall, we observe $5.1\times$ speedup on average over the PyTorch baseline. This result shows significant performance benefits from utilizing Faith on accelerating transformer verification. Comparing across lengths, we observe a higher speedup of $5.54\times$ over the PyTorch baseline on shorter sentences with 2 to 32 words. The reason is that our autotuning optimization (§5) automatically adjusts the number of threads and memory layout to improve the parallelism. We achieve a smaller speedup of $3.85\times$ on longer sentences with 64 and 128 words. For these longer sentences, we have achieved high occupancy on GPUs and the speedup is limited by the hardware capability.

Surprisingly, we observe that TVM and Anso achieve $0.33\times$ and $0.73\times$ speedup, which is significantly slower than PyTorch baselines on verification of matrix multiplication. The main reason is that TVM and Anso focus on accelerating standard NNs and cannot efficiently support computing

patterns in the verification of matrix multiplication (Fig. 3(c)). Instead, Faith exploits a semantic-aware kernel fusion (§3.1) to efficiently support such computing patterns in verification.

We show speedup on verification of matrix multiplication over the diverse embedding sizes in Fig. 11. We study embedding size from 64 to 640 following popular transformer settings. We note that transformer in natural language processing usually adopts a relatively small embedding size (e.g., 64 to 256), which is different from convolutional neural networks in computer vision that adopts a large embedding size (e.g., 1024). Overall, Faith achieves $4.2\times$ speedup on average over the PyTorch baseline. This result shows that Faith can improve performance over diverse embedding sizes. We also observe that Faith achieves larger speedup for smaller embedding sizes, which is similar to the case when verifying matrix multiplication over diverse lengths.

Performance benefits on verification of ReLU. We show speedup on verification of ReLU over diverse lengths in Fig. 12. As we discussed earlier in §4.1, verification of ReLU represents an important computing pattern of verifying elementwise operators. Due to similar behaviors between diverse lengths and embedding sizes, we focus on verification over diverse lengths and keep embedding size as 128, which is a popular setting in transformers. Overall, Faith achieves $141\times$ speedup over PyTorch baseline. This large speedup shows it promising to accelerate verification of elementwise operators. Besides, Faith achieves $13.4\times$ and $13.5\times$ speedup over TVM and Anso. The reason is that our *workload-adaptive reduction* (§4.2) can significantly improve parallelism during reduction and *sharing-oriented workload sharing* can minimize memory access with GPU memory hierarchy.

#Layers	1	2	3	4	5	6
PyTorch	9.1	18	25	28	31	37
Faith	4	7.2	7.8	10.9	12.6	15.4

Table 3: Latency on SST dataset and A100. Unit: Second.

Performance benefits on verifying Tanh and dot product layers. We show the speedup from Faith over the PyTorch baseline on verification of Tanh and verification of dot product in Fig. 13. We skip the results of TVM and Ansor since these two frameworks do not support computing patterns in verification of Tanh and verification of dot product. Here, we show results of verification of Tanh since it is a popular elementwise operator in transformer verification. We also show results of verification of dot product since it accounts for around 45% latency in transformer verification. Overall, we observe that Faith achieves $138\times$ speedup on average for verification of Tanh. This result is similar to the performance improvement for verification of ReLU, since both Tanh and ReLU are elementwise operators and share benefits from the same set of optimizations. We also observe that Faith achieves $26.5\times$ speedup on average for verification of dot product. This result shows the performance benefits from semantic-aware kernel fusion (§3.1) and broadcast-aware super threading (§4.4) that mitigate redundant memory access.

Raw latency on transformer verification. We show the raw latency for transformer verification on the SST dataset and NVIDIA A100 GPU in Table 3. Faith requires only a few seconds to verify the NN prediction on a long sentence (with on average 25 tokens). More specifically, when verifying transformers with 1 to 6 layers, Faith only requires 4 to 15.4 seconds to verifying a sentence. This results brings transformer verification to the level of being practical for use.

7 Discussion

Why Faith performs better than prior approaches. Existing frameworks, such as PyTorch, TVM, and Ansor, only support limited computation patterns for standard NNs. They cannot directly support bound-centric computation patterns in transformer verification. While several frameworks like TVM allow autotuning for diverse operators, there is no magic. They still rely on hand-written GPU kernels (e.g., matrix multiplication) as the parametric templates (e.g., with tiling size as a parameter) and can only tune these tiling sizes. When applying to bound-centric computation patterns, they will break an operator for transformer verification into several hand-written GPU kernels for standard NNs. This leads to significantly higher memory access when aggregating computation results across GPU kernels into one transformer verification output.

Instead, Faith provides direct support for bound-centric computation patterns. Instead of breaking into several GPU kernels for standard NNs, we consider the bound-centric computation patterns as a whole and design a set of optimizations to reduce the memory and computation cost. For example,

we found the lower and upper bounds are usually multiplied with the same weight matrix and can be loaded once to reduce memory overhead.

Practicality of transformer verification with Faith. Faith brings transformer verification to be practical by consuming only around 10 seconds to verify a long sentence (e.g., 25 tokens). We remark that transformer verification is one of the hottest topics in deep learning. Hundreds of related papers have been published in top deep learning conferences. The performance is essential to bring transformer verification into practical applications. However, existing efforts mainly reside in the algorithmic domain. In this paper, we build the first framework for efficient transformer verification on GPUs. Our work will open a new system research direction on developing high-performance systems for deep learning verification.

8 Conclusion

Verifying the robustness of transformers draws increasing attention from both the academic and industry fields over the recent years. Unfortunately, an efficient system design for transformer verification is still yet to come. Existing transformer verification still exploits standard neural network frameworks which are unoptimized towards transformer verification workload. The main reason is that efficient systems for transformer verification require both expertise from the machine learning community on mathematical verification designs and the system community on efficient memory and parallelism designs.

In this paper, we propose a Faith framework for efficient transformer verification. Specifically, we first design a set of semantic-aware computation graph transformations to fully exploit fusion opportunities in transformer verification at the computation graph level. Then, we propose a verifier-specialized kernel crafter to efficiently map fused verification kernels towards modern GPUs with minimized memory overhead and improved parallelism. Finally, we propose an expert-guided autotuning to dynamically optimize kernels according to the transformer verification workload and GPU backend characteristics. Comprehensive experimental evaluation shows that Faith significantly improves the performance of transformer verification over state-of-the-art frameworks.

Looking ahead, we believe our work in transformer verification would highlight a new direction on developing high-performance systems for deep learning verification. This will encourage system experts with diverse backgrounds to build the next-generation deep learning systems and facilitate the wide application of secure deep learning.

References

- [1] Nader Akoury, Kalpesh Krishna, and Mohit Iyyer. Syntactically supervised transformers for faster neural machine translation. In *ACL (1)*, pages 1269–1281. Association for Computational Linguistics, 2019.

- [2] Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani B. Srivastava, and Kai-Wei Chang. Generating natural language adversarial examples. In *EMNLP*, pages 2890–2896. Association for Computational Linguistics, 2018.
- [3] Melika Behjati, Seyed-Mohsen Moosavi-Dezfooli, Mahdieh Soleymani Baghshah, and Pascal Frossard. Universal adversarial attacks on text classifiers. In *ICASSP*, pages 7345–7349. IEEE, 2019.
- [4] Gregory Bonaert, Dimitar I. Dimitrov, Maximilian Baader, and Martin T. Vechev. Fast and precise certification of transformers. In *PLDI*, pages 466–481. ACM, 2021.
- [5] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 785–794. ACM, 2016.
- [6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *OSDI*, pages 578–594. USENIX Association, 2018.
- [7] Junyan Cheng, Iordanis Fostiropoulos, Barry W. Boehm, and Mohammad Soleymani. Multimodal phased transformer for sentiment analysis. In *EMNLP (1)*, pages 2447–2458. Association for Computational Linguistics, 2021.
- [8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, pages 4171–4186. Association for Computational Linguistics, 2019.
- [9] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*. OpenReview.net, 2021.
- [10] Facebook. How facebook uses super-efficient ai models to detect hate speech. <https://ai.facebook.com/blog/how-facebook-uses-super-efficient-ai-models-to-detect-hate-speech/>.
- [11] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbo Transformers: an efficient GPU serving system for transformer models. In Jaejin Lee and Erez Petrank, editors, *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, pages 389–402. ACM, 2021.
- [12] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. EGEMM-TC: accelerating scientific computing on tensor cores with extended precision. In *PPoPP*, pages 278–291. ACM, 2021.
- [13] Boyuan Feng, Yuke Wang, Tong Geng, Ang Li, and Yufei Ding. APNN-TC: accelerating arbitrary precision neural networks on ampere GPU tensor cores. In Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin, editors, *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, pages 37:1–37:13. ACM, 2021.
- [14] Siddhant Garg, Thuy Vu, and Alessandro Moschitti. Tanda: Transfer and adapt pre-trained transformer models for answer sentence selection. <https://www.amazon.science/publications/tanda-transfer-and-adapt-pre-trained-transformer-models-for-answer-sentence-selection>.
- [15] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR (Poster)*, 2015.
- [16] Yu-Lun Hsieh, Minhao Cheng, Da-Cheng Juan, Wei Wei, Wen-Lian Hsu, and Cho-Jui Hsieh. On the robustness of self-attentive models. In *ACL (1)*, pages 1520–1529. Association for Computational Linguistics, 2019.
- [17] Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter Szolovits. Is BERT really robust? A strong baseline for natural language attack on text classification and entailment. In *AAAI*, pages 8018–8025. AAAI Press, 2020.
- [18] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *CAV (1)*, volume 10426 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2017.
- [19] Bumsoo Kim, Junhyun Lee, Jaewoo Kang, Eun-Sol Kim, and Hyunwoo J. Kim. HOTR: end-to-end human-object interaction detection with transformers. In *CVPR*, pages 74–83. Computer Vision Foundation / IEEE, 2021.

- [20] Junjie Lai and André Seznec. Performance upper bound analysis and optimization of SGEMM on fermi and kepler gpus. In *CGO*, pages 4:1–4:10. IEEE Computer Society, 2013.
- [21] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. In *ICLR*. OpenReview.net, 2021.
- [22] Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. Textbugger: Generating adversarial text against real-world applications. In *NDSS*. The Internet Society, 2019.
- [23] Xin-Chun Li, De-Chuan Zhan, Jia-Qi Yang, and Yi Shi. Deep multiple instance selection. *Sci. China Inf. Sci.*, 64(3), 2021.
- [24] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. A coordinated tiling and batching framework for efficient GEMM on gpu. In *PPoPP*, pages 229–241. ACM, 2019.
- [25] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.
- [26] Yu Lu, Jiali Zeng, Jiajun Zhang, Shuangzhi Wu, and Mu Li. Attention calibration for transformer in neural machine translation. In *ACL/IJCNLP (1)*, pages 1288–1298. Association for Computational Linguistics, 2021.
- [27] Paulius Micikevicius. Local memory and register spilling. https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf.
- [28] Tomáš Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.
- [29] Nvidia. Cuda c++ programming guide. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- [30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8024–8035, 2019.
- [31] Lihua Qian, Hao Zhou, Yu Bao, Mingxuan Wang, Lin Qiu, Weinan Zhang, Yong Yu, and Lei Li. Glancing transformer for non-autoregressive neural machine translation. In *ACL/IJCNLP (1)*, pages 1993–2003. Association for Computational Linguistics, 2021.
- [32] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.
- [33] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.
- [34] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: a new IR for machine learning frameworks. In *MAPL@PLDI*, pages 58–68. ACM, 2018.
- [35] Zhouxing Shi, Huan Zhang, Kai-Wei Chang, Minlie Huang, and Cho-Jui Hsieh. Robustness verification for transformers. In *ICLR*. OpenReview.net, 2020.
- [36] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, pages 1631–1642. ACL, 2013.
- [37] Hao Tang, Donghong Ji, Chenliang Li, and Qiji Zhou. Dependency graph enhanced dual-transformer structure for aspect-based sentiment classification. In *ACL*, pages 6578–6588. Association for Computational Linguistics, 2020.
- [38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.
- [39] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: dynamic GPU memory management for training deep neural networks. In Andreas Krall and Thomas R. Gross, editors, *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2018, Vienna, Austria, February 24-28, 2018*, pages 41–53. ACM, 2018.
- [40] Yuke Wang, Boyuan Feng, and Yufei Ding. QGTC: accelerating quantized graph neural networks via GPU tensor core. In Jaejin Lee, Kunal Agrawal, and Michael F.

- Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seoul, Republic of Korea, April 2 - 6, 2022, pages 107–119. ACM, 2022.
- [41] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An adaptive and efficient runtime system for GNN acceleration on gpus. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 515–531. USENIX Association, 2021.
- [42] Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. Automatic perturbation analysis for scalable certified robustness and beyond. In *NeurIPS*, 2020.
- [43] Da Yan, Wei Wang, and Xiaowen Chu. Optimizing batched winograd convolution on gpus. In *PPoPP*, pages 32–44. ACM, 2020.
- [44] Fuzhi Yang, Huan Yang, Jianlong Fu, Hongtao Lu, and Baining Guo. Learning texture transformer network for image super-resolution. In *CVPR*, pages 5790–5799. Computer Vision Foundation / IEEE, 2020.
- [45] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *NeurIPS*, pages 5754–5764, 2019.
- [46] Han-Jia Ye, Yi Shi, and De-Chuan Zhan. Identifying ambiguous similarity conditions via semantic matching. In *CVPR*. Computer Vision Foundation / IEEE, 2022.
- [47] Linwei Ye, Mrigank Rochan, Zhi Liu, and Yang Wang. Cross-modal self-attention network for referring image segmentation. In *CVPR*, pages 10502–10511, 2019.
- [48] Da Yin, Tao Meng, and Kai-Wei Chang. Sentibert: A transferable transformer-based architecture for compositional sentiment semantics. In *ACL*, pages 3695–3706. Association for Computational Linguistics, 2020.
- [49] Feng Zhang, Zaifeng Pan, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. G-TADOC: enabling efficient gpu-based text analytics without decompression. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 1679–1690. IEEE, 2021.
- [50] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. Poelib: A high-performance framework for enabling near orthogonal processing on compression. *IEEE Trans. Parallel Distributed Syst.*, 33(2):459–475, 2022.
- [51] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. TADOC: text analytics directly on compression. *VLDB J.*, 30(2):163–188, 2021.
- [52] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. In *NeurIPS*, pages 4944–4953, 2018.
- [53] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *NIPS*, pages 649–657, 2015.
- [54] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *OSDI*, pages 863–879. USENIX Association, 2020.
- [55] Xizhou Zhu, Weijie Su, Lewei Lu, Bin Li, Xiaogang Wang, and Jifeng Dai. Deformable DETR: deformable transformers for end-to-end object detection. In *ICLR*. OpenReview.net, 2021.

DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs

Weihao Cui*, Han Zhao*, Quan Chen*, Hao Wei*, Zirui Li*, Deze Zeng[◇], Chao Li*, Minyi Guo*
*Shanghai Jiao Tong University, [◇]China University of Geosciences

Abstract

The DNN inferences are often batched for better utilizing the hardware in existing DNN serving systems. However, DNN serving exhibits diversity in many aspects, such as input, operator, and load. The unawareness of these diversities results in inefficient processing. Our investigation shows that the inefficiency roots in the feature of the existing batching mechanism: one entry and one exit. Therefore, we propose **DVABatch**, a runtime batching system that enables the multi-entry multi-exit batching scheme. We first abstract three meta operations, new, stretch, and split, for adjusting the ongoing batch of queries to achieve the multi-entry multi-exit scheme. The meta operations could be used to form different scheduling logics for different diversities. To deliver the meta operations to an ongoing batch, we slice the DNN models into multiple stages. Each stage corresponds to one executor, which is managed by a state transition diagram. Compared with state-of-the-art solutions, our experimental results show that DVABatch reduces 46.4% average latency and achieves up to $2.12\times$ throughput improvement.

1 Introduction

Deep neural networks (DNNs) [27, 37, 57] are widely used in intelligent services [1, 5, 6, 12]. Since user queries have stringent QoS in terms of end-to-end latency, dedicated accelerators like GPUs [10, 11] and NPUs [21] are used to speed up the DNN inferences. However, a single DNN query often cannot fully utilize these accelerators [17, 18, 43, 46, 65] (e.g., An Nvidia Titan RTX GPU has 72 SMs [10]). Therefore, emerging DNN serving systems (e.g., Clipper, Triton, TF-Serving) [9, 23–26, 51, 62, 63] batch queries for better taking advantage of the accelerators' parallelism. Queries that arrive in a given batch time window are organized into a batch, and an *executor* (process) is used by the DNN serving system to process the entire batch at a time. Such a batching policy uses the same batch size (*bs*) across a single inference process.

On GPUs, due to the single program multiple-data (SPMD) design, all the queries in a batch return at the same time. This

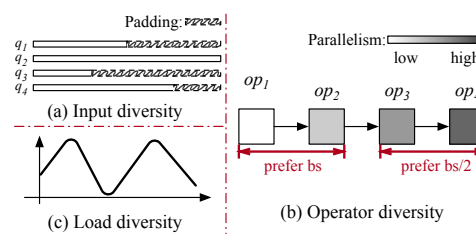


Figure 1: Serving diversities in real-world services.

batching pattern is referred to be the *single-entry single-exit batching pattern*. It works great for best-effort applications, and the services when the queries arrive in uniform intervals [22]. However, DNN serving scenarios show various diversities, and the single-entry single-exit batching pattern results in the long response latency of inference queries on GPUs. For instance, we find at least three types of diversities when serving DNN models, as shown in Figure 1.

Input Diversity. The inputs of user queries show high diversity (e.g., in natural language processing services). Short queries are all padded to the size of the longest query for batching. The benefit of batching may be negated by the wasted computation of the padded part.

Operator Diversity. While all the operators of a DNN model share the same batch size, they have different preferred batch sizes. An operator's preferred batch size is the smallest batch size that fully utilizes the current GPU. The hardware is not fully utilized if an operator's preferred batch size is larger than the used batch size. Otherwise, the processing time is increased unnecessarily.

Load Diversity. The service queries do not arrive in a uniform interval. In this case, the number of queries collected in a single batch time window varies. When the load bursts, a previous non-full batch results in the long latency of subsequent queries. In other words, hardware resources are wasted while the queries are waiting in the next batch time window.

The diversities result in inefficient processing of user queries (discussed in more detail in Section 3). The ineffi-

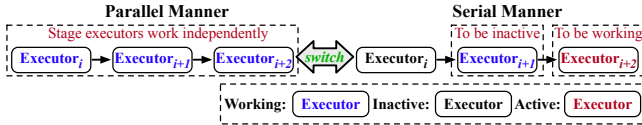


Figure 2: The work manner switch of stage executors.

ciency stems from the batching pattern of single-entry single-exit. To address the inefficiency, we, therefore, propose a *multi-entry multi-exit batching scheme* for DNN serving on GPUs. For instance, with the multi-entry multi-exit batching scheme, a short query can exit early without waiting for the entire batch to exit (input diversity), a batch can be split into smaller batches to execute an operator with a preferred smaller batch size (operator diversity), and the queries that arrive later can join an ongoing but non-full batch (load diversity).

It is nontrivial to implement the multi-entry multi-exit batching scheme for GPUs. *It necessitates the ability of the serving system to dynamically alter the batch size of an ongoing query batch.* Specifically, such a system should enable the joining of incoming queries into the ongoing batch and the splitting of an ongoing batch into smaller batches (The queries in the smaller batches could exit independently). *Moreover, it introduces extra complexity for designing executors in the DNN serving system.* With the multi-entry multi-exit scheme, the inference of batched queries is broken down into multiple stages, and each stage’s execution requires one executor. Multiple executors have to coordinate with each other to ensure the validity of the query inference.

To this end, we propose **DVABatch** to enable the multi-entry multi-exit batching scheme effectively. DVABatch provides three meta operations, *new*, *stretch*, and *split*, to adjust the ongoing batch (Section 5). The *new* operation creates a new batch, just like the traditional batching strategy. The *stretch* operation adds new queries to the ongoing batch. The *split* operation breaks a running batch into multiple batches, which could be scheduled separately. Query batching can be done in a variety of ways using the three meta operations.

To deliver the meta operations to the stage executors, a *batch queue* that stores the batch information is added between adjacent stage executors, and a global *batch table* is utilized to record the to-be-performed meta operations at each stage (Section 5). When an executor completes its computation for a batch of queries, it verifies the to-be-performed meta operations for the next stage in the *batch table*. If a *split* or *stretch* operation is required, the executor applies the corresponding meta operation on the current batch and pushes new batches of queries into the batch queue of the next stage.

While multiple active executors run independently like a software pipeline, DVABatch should manage them properly. Otherwise, the naive parallel execution of the executors invalidates the execution due to data hazard and results in unnecessary long latency. For instance, the executor should

run batches with different input sizes in parallel for the input diversity, and run the sub batches after the *split* meta operation sequentially for the operator diversity. DVABatch introduces a state transition diagram based solution to support the executors’ complicated runtime scheduling (Section 6). Each executor has four states: *active*, *checking*, *working*, and *inactive*. Through the state transition diagram, the work manners depicted in Figure 2 are both supported.

The main contributions of this paper are as follows.

- We propose a multi-entry multi-exit batching scheme for efficient DNN service processing on GPUs.
- We provide a general scheduling mechanism that leverages meta operations, and state transition diagram to create policies for different serving diversities.
- We implement DVABatch with Triton, a state-of-the-art DNN serving system. Our experimental results on Nvidia Titan RTX show that DVABatch reduces 46.4% average latency and achieves up to $2.12\times$ throughput improvement for the involved serving diversities.

2 Related Work

Many systems have been proposed for efficient DNN inference [9, 25, 35]. Clipper [23], TF-Serving [51], Triton [9] adopted the traditional batch strategy that uses batch time window and the maximum allowed batch size. They treated the DNN model as an indivisible whole. They left the scheduling of inner operators to their supported backends. These works do not perceive the serving diversity and utilize the DNN operator scheduling for efficient processing.

There are some prior research on improving operator scheduling. TensorFlow Fold [47], DyNet [49], and BatchMaker [31] focused on the runtime scheduling of operators for RNN. They are model-specific solutions, removing padding for RNNs. The RNN cells of the same type share the same parameter weights and are executed recursively [38]. These works relied on this property to remove the padding. The design is restricted to resolving the input diversity for RNN. It cannot be applied to other models with input diversity, e.g., attention-based models, and BatchMaker-like batch mechanism can be achieved through DVABatch’s meta operations. Besides, LazyBatch [22] cared about the load fluctuation and proposed batching queries lazily. LazyBatch performed per-operator scheduling that incurs high scheduling overhead. It could not handle other diversities. In this work, we focus on resolving the problems caused by serving diversities in a holistic way.

There are also some prior works about ragged tensor for the input diversity [4, 29]. They generated the customized implementation of operators to remove the padding. However, operators like GEMM and convolution cannot be optimized through these works, which dominate the computation in DNN. These works are orthogonal to DVABatch and can be combined together to enable even lower latency.

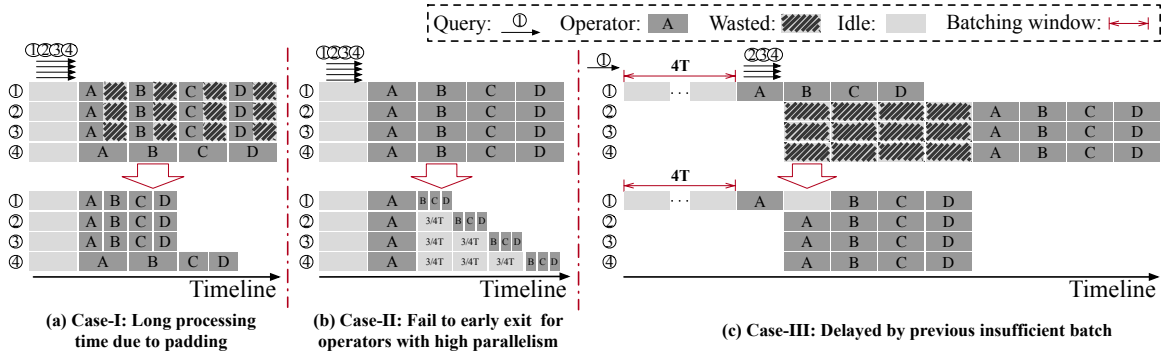


Figure 3: The long latency of user queries due to (Case-I) input diversity, (Case-II) operator diversity, and (Case-III) load diversity.

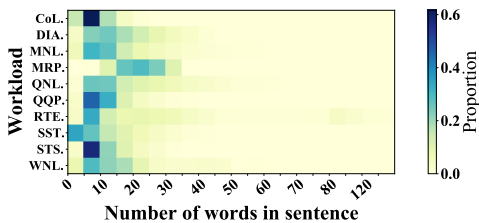


Figure 4: The sequence distribution of workloads in GLUE.

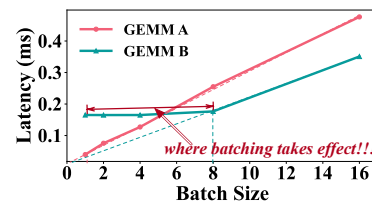


Figure 5: Latencies of two GEMM operators with different batch sizes on Titan RTX.

3 Background and Motivation

This section shows the long query latency problem due to the single-entry single-exit batching, and motivates the design of DVABatch. Figure 3 shows the three involved diversities. For simplicity of illustration, we assume that each operator completes in 1 time unit (T), and the batch size is 4. In this case, once 4 queries are received or the batch time window ends, the received queries are batched and issued to run.

3.1 Input diversity

Input diversity widely exists in DNN services. E.g., natural language processing services often process sentences of different lengths. Figure 4 shows the sequence length distribution in 10 workloads of the General Language Understanding Evaluation (GLUE) dataset [59]. As observed, most sentences have 5-20 words, but some have more than 100 words.

For these models, the input of short queries is padded to the same size as the input of the longest query so that they can be batched to run [28]. Case-I in Figure 3 shows the batching with input diversity. As shown in the upper part of Case-I, the hardware resources are wasted for the computation of extra paddings (the queries in a batch return simultaneously).

The lower part of Case-I shows a better batching strategy: the batch is divided into two smaller batches, one for short queries ①, ②, and ③, and one for the long query ④. In this way, queries ①, ②, and ③ return earlier, and the average latency is reduced by 37.5% (from $4T$ to $2.5T$). Note that the two

batches may run in parallel before the short batch completes if 4 operators are required to fully utilize the GPU.

3.2 Operator diversity

For a DNN service, the operators often require different batch sizes to fully utilize the GPU. Figure 5 shows the latencies of two General Matrix Multiplication (GEMM) operators converted from two convolution operators of Resnet50 [37], with the shapes of $[bs * 3136, 576] \times [576, 64]$ and $[bs * 49, 576] \times [576, 512]$. GEMM operators dominate DNNs (occupying 86% of the computation time) [44].

As shown, the preferred batch sizes of *GEMM-A* and *GEMM-B* are 1 and 8, respectively. For *GEMM-A*, batching only increases its latency without improving the processing throughput. For *GEMM-B*, using a batch size smaller than 8 is not able to fully utilize a GPU (the processing time does not increase until the batch size is larger than 8).

Case-II in Figure 3 shows the batching with operator diversity. In Case-II, operator *A* prefers batch size 4, operator *B*, *C*, and *D* prefer batch size 1. The lower part of Case-II shows a better batching strategy: we can run operator *A* with batch size 4, split the batch into four smaller batches with batch size 1 at operator *B*, and run the small batches sequentially. In this way, query ①, ②, and ③ return earlier. The average latency can be reduced by 28.1% (from $4T$ to $2.875T$).

Many DNN models suffer from operator diversity. Figure 6 shows the processing time of the operators in Unet [56], a widely used image segmentation network with different batch

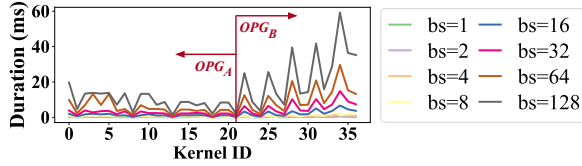


Figure 6: Operator duration of Unet on Titan RTX with variable batch sizes.

sizes on an Nvidia Titan RTX GPU. In the figure, the x-axis represents each operator’s id in the executive order. As observed, most operators in the former part (OPG_A) benefit from large batch sizes (e.g., larger than 32), but the operators in the latter part (OPG_B) only benefit from small batch sizes (smaller than 8). Using a batch size larger than 8 for Unet, the operators in OPG_B have longer latency without throughput improvement. On the contrary, using a batch size smaller than 32, the operators in OPG_A do not fully utilize the GPU.

3.3 Load diversity

User queries do not arrive in a uniform interval, as end users may randomly submit their queries. The number of received queries in a single batch time window varies [30,34,36,41,55].

Case-III in Figure 3 presents the batching with the load diversity. The batch time window is $4T$, the operators prefer batch size 4. With the current batching policy (the upper part of Case-III), query ① starts to run alone after it waits for $4T$, and the GPU is not fully utilized. During the processing of query ①, three queries ②, ③, and ④ arrive, but they have to wait to be executed in the next batch.

The lower part of Case-III shows a better way to run the four queries: the first batch (query ①) waits for the second batch after the first operator A completes. Then, the two insufficient batches are merged into a new batch that fully utilizes the hardware. In this way, the average latency can be reduced by 34.4% (from $8T$ to $5.25T$).

3.4 Diversities among DNN services

The three types of diversities may exist in the same DNN services. Current batching policies with simple modifications cannot effectively handle them. In this case, designing a static batching policy is not able to fulfill the ever-changing diversities. A low-level batching mechanism that supports configuring the batching policy accordingly is required. Analyzing the three better batching cases in Figure 3, they share three requirements for the batching mechanism.

First of all, the mechanism should be able to interrupt an ongoing batch so that we can adjust the inappropriate batching decision. Second, the mechanism should be able to split a large batch into small batches. In this way, the small batches may run in a parallel manner (Case-I) or sequentially (Case-

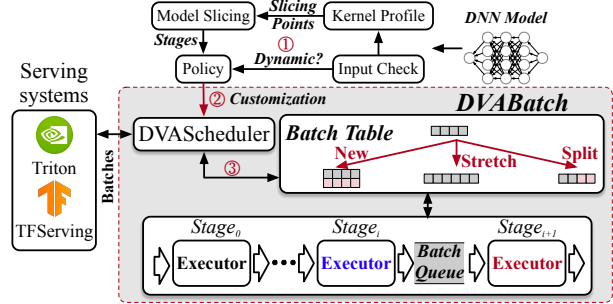


Figure 7: Design of DVABatch.

II). Third, the mechanism should be able to merge multiple insufficient batches. In this way, we can build a large batch to better utilize the hardware resource (Case-III).

A multi-entry multi-exit batching scheme fulfills all three requirements, and has the potential to achieve better batching, together with appropriate batching policies.

4 Design of DVABatch

We therefore design and propose DVABatch to resolve the long latency problem due to the serving diversities.

4.1 Overview

Figure 7 illustrates DVABatch’s methodology. DVABatch enables the multi-entry multi-exit batching scheme for the upper-level DNN serving systems (e.g., Triton, TF-Serving).

In general, in order to support the multi-entry and multi-exit batching, DVABatch slices a DNN model into multiple fine-grained stages, and each stage has multiple adjacent operators. The queries are able to join a batch at the beginning of a stage, and exit from a batch at the end of a stage. Based on the stages, DVABatch designs and implements batching policies that manage the batching operation of each stage, based on the real-time diversities. DVABatch supports three meta-operations *new*, *stretch*, *split* for adjusting the batching. The *new* operation creates a new batch, *stretch* adds new queries to the ongoing batch, and *split* breaks a running batch into multiple smaller batches. Various batching policies can be implemented based on the meta-operations.

As shown in Figure 7, DVABatch comprises a *batch table*, *stage executors*, *batch queues*, and *DVAScheduler*.

- The *batch table* records the running status of the ongoing batches. It supports three meta operations for adjusting the batches at each stage.
- A *stage executor* is a process that is responsible for the corresponding stage’s execution. DVABatch utilizes a state transition diagram for executor management.
- There is a *batch queue* between the two adjacent stages, for transmitting the batch information. A stage executor pulls

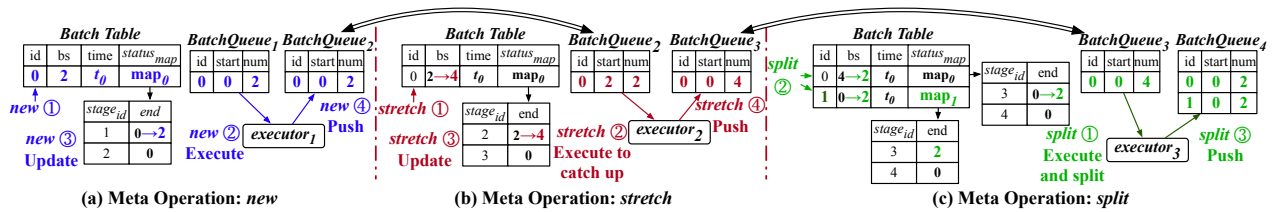


Figure 8: Implementing the meta operations with batch table and batch queues between stages.

out batches from its previous *batch queue* for execution, and pushes batches into the *batch queue* of the next stage.

- *DVAScheduler* provides diversity-aware scheduling using various batching policies implemented with the three meta operations. The policies can be customized according to the serving diversity.

4.2 The Serving Workflow with DVABatch

Figure 7 also shows DVABatch’s serving workflow for the involved diversities. The steps for using DVABatch to serve a DNN service are as follows:

- ① DVABatch checks the input data pattern of the service, and profiles it to obtain the diversity patterns. Based on the profiling, the DNN model is sliced into stages automatically. A diversity-aware policy is generated for the DNN model.
- ② Each stage executor loads the corresponding stage of the model, and *DVAScheduler* uses the scheduling logic in the policy to schedule the accepted queries.
- ③ If a specific condition defined in the *DVAScheduler* is satisfied, the batch table is instructed to adjust the ongoing batch by *new*, *stretch*, *split* operations accordingly.

For a new DNN service, DVABatch handles input diversity before operator diversity. This is because handling input diversity directly reduces computation while handling operator diversity better schedules computation.

5 Enabling Multi-entry Multi-exit Scheme

In this section, we propose the abstraction of meta operations and how we achieve the multi-entry multi-exit scheme.

5.1 Defining the Meta Operations

As stated before, to achieve low latency query scheduling, the batched queries should be able to join and exit the batching system in several forms: **Multi-entry**. When a new batch of queries arrives, it can interrupt an ongoing batch, catch up with the progress of the interrupted one, then be merged into a single larger batch. In addition, the new incoming batch also can join the processing by co-running with the ongoing batches in different stage executors without pausing the ongoing one. **Multi-exit**. If some queries inside a batch need to

exit early, we need to split the batch into several batches and allow them to exit execution independently.

DVABatch abstracts three meta scheduling operations inside the *DVAScheduler*: *new*, *stretch*, and *split*, for supporting the multi-entry multi-exit scheme. With *new*, the new incoming queries are organized into a new batch. The batch created by *new* operation could co-run with the previous batches; With *stretch*, an ongoing batch is stretched with new incoming queries. At a specific stage, these queries are merged into the ongoing batch for processing; With *split*, a large ongoing batch is split into several smaller batches to be processed separately. The three meta operations can be used to form complicated scheduling logic when necessary.

5.2 Implementing the Meta Operations

It is challenging to support the meta operations, as a batch runs in multiple stages. The meta operations should be performed based on the stages’ execution status. In general, DVABatch tracks the stage status of the batches, and notifies the meta operations to the corresponding stage executors.

5.2.1 Batch Table and Batch Queues

DVABatch uses a *batch table* to track the processing status of all the batches on a GPU. The batch table is updated by the *DVAScheduler* through the meta operations.

As shown in Figure 8, a row in the batch table records the status of an ongoing batch. In a row, *id* is the batch’s identifier, *bs* is its current batch size, *time* is the timestamp the batch is created, *status_map* is a map that records the number of completed queries in each stage of the batch. For instance, the first row of *status_map* in Figure 8(a) means stage 1 has completed 2 queries in its batch. We need *status_map* because the latter queries of a *stretched* batch should catch up with the ongoing queries. With *status_map*, the executors could get the right number of queries to execute after *stretch* operation.

After the current stage executor completes its execution, it notifies the subsequent stage executor to run. DVABatch maintains *batch queues* between adjacent stage executors to trigger such execution. In a row of a batch queue, *id* is the batch’s identifier, *start* is the id of the to-be-processed query, and *num* is the number of to-be-processed queries in the batch.

The stage executor pulls a batch from its batch queue, and processes the queries accordingly. For instance, $executor_1$ in Figure 8(a) will run query 0 to query 1 ($start = 0, num = 2$) in the current batch. Once the execution completes, the stage executor updates the row of the processed batch in batch table, and pushes an item into the next batch queue.

5.2.2 Handling Meta Operations

Based on batch table and batch queues, Figure 8 shows an example that three meta operations are performed on the same batch, $batch_0$. In the example, we first *new* the batch $batch_0$ with 2 queries (Figure 8(a)); Then, we *stretch* the batch $batch_0$ with another 2 new queries while it is already processed by $executor_2$ (Figure 8(b)); Last, we *split* the batch $batch_0$ into 2 smaller batches at the third stage (Figure 8(c)).

Handling new. Once $batch_0$ is received, ① a *new* operation is instructed, and a new item is added to the batch table. Meanwhile, an item is pushed to the first stage executor’s batch queue ($BatchQueue_1$). ② the executor of the first stage (hereinafter, we refer to the executor of $stage_i$ as $executor_i$) is notified to obtain the item and perform the execution. Once $executor_1$ completes, it ③ updates $status_{map}$ in the batch table, and ④ pushes an item into $BatchQueue_2$.

Handling stretch. ① As 2 new queries are added into $batch_0$ with the *stretch* operation at stage 2, bs of $batch_0$ in the batch table is changed from 2 to 4. Because $batch_0$ is *stretched* to 4 queries while being processed by $executor_2$, the executor does not push an item into $BatchQueue_3$, but only updates $status_{map}$. ② A new item (a batch with $id = 0, start = 2, num = 2$) is pushed into $BatchQueue_1$, so that the newly added queries can catch up with the progress of the current batch. ③ Once the new queries catch up, $executor_2$ updates $status_{map}$ and ④ pushes a merged batch into $BatchQueue_3$ (a batch with $start = 0$ and $num = 4$). The *stretch* operation is only performed on the latest batch stored in the batch table.

Handling split. When $batch_0$ goes to stage 3, ① $executor_3$ pulls $batch_0$ from the batch queue and runs it with $bs = 4$. After that, $executor_3$ is instructed with a *split* operation. ② The original batch is split into two batches in the batch table. ③ Lastly, the generated batches are all pushed into $BatchQueue_4$ one by one. The *split* operation can happen when a new batch of queries comes or during the execution of an ongoing batch.

The meta operations co-exist without conflict. Although *split* happens at any stage, a potential *split* operation can be known when generating the batch by *new* and *stretch*. We disable further *stretch* for the batch that we perceive *split* operation. The batches generated by *split* inherit this property.

6 Managing Stage Executors

In this section, we present the way the stage executors are organized to process the batches.

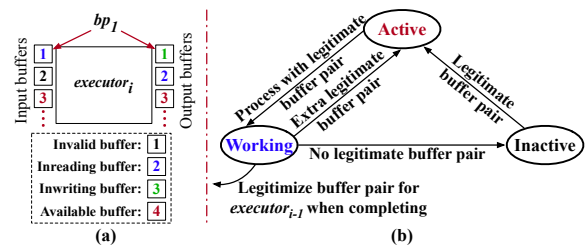


Figure 9: (a) Stage executor processes batches with multiple buffers; (b) Traditional state transition diagram of $executor_i$.

6.1 Processing with Multiple Buffers

With multiple stages, the executors of adjacent stages have “producer-consumer” relationships. In this case, there are data hazards on the buffers between stages.

Figure 9(a) shows the way the stages are connected. Multiple buffers are used because multiple batches may be active concurrently. An executor needs to obtain an input-output buffer pair before it can process a batch. The output buffer of $executor_i$ is also the input buffer of $executor_{i+1}$. If $executor_i$ is using a buffer pair bp , there is a Write-After-Read hazard on bp ’s input buffer, as $executor_{i-1}$ may write to the very buffer before $executor_i$ reads the data. Similarly, there is a Read-After-Write hazard on bp ’s output buffer.

For ensuring the execution correctness, a buffer can be in the *available*, *invalid*, *inreading*, or *inwriting* state. A buffer is *invalid* when it cannot be used as an input buffer currently. It is *inreading/inwriting* when it is used as an input/output buffer for a batch’s execution. It is *available* when it is not used by any executor. Figure 9(a) shows an example that $executor_i$ is using the first buffer pair bp_1 , and $executor_{i+1}$ is using the output buffer of the second buffer pair bp_2 as its input.

- The input buffer of bp_1 is in *inreading* state and the output buffer of bp_1 is in *inwriting* state.
- The input buffer of bp_2 is in *invalid* state because the output buffer of bp_2 is currently used by $executor_{i+1}$ for execution. $executor_i$ cannot use it to run a new batch.
- The third buffer pair are both in *available* states.

A stage executor can run a batch only when it successfully obtains a legitimate buffer pair. A buffer pair is legitimate, when both the input and output buffer are *available*, or the input buffer is *available* but the output buffer is *invalid*. This is because an *invalid* buffer can be used as the output buffer of an executor, but cannot be used as the input buffer of the later stages.

6.2 State Transition of the Executors

Based on the buffer states, there are some traditional ways to create a naive state transition rule for the stage executors to run as a pipeline. For instance, a stage executor can be in three states: *active*, *working*, *inactive*, and the states change

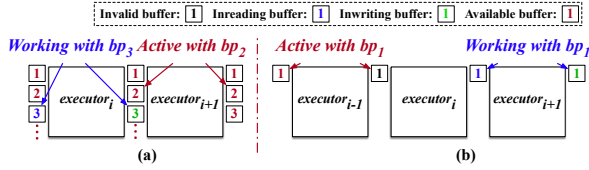


Figure 10: (a) Inconsistency of buffer pairs; (b) Fail to run in serial manner with single buffer pair.

according to the diagram in Figure 9(b).

However, these traditional state transition rules assume all the stage executors use the buffer pairs in some fixed order (e.g., ID order). **While the traditional rules work well for the single-entry single-exit pipeline, they will encounter the validity problem for the multi-entry multi-exit pipeline.**

Specifically, the requirement of meta operations *stretch* and *split* invalids the above traditional transition rules. *stretch* merges the outputs from different buffer pairs into a single one, and *split* may split the output into multiple buffer pairs. That means some stage executors may use more buffer pairs than others, and the access order of these buffer pairs is scrambled. As shown in Figure 10(a), while $executor_i$ is using bp_3 for execution, $executor_i$ is active with bp_2 . Such inconsistency invalidates the traditional state transition rule.

On the other hand, stage executors need to run multiple batches in parallel for load diversity and input diversity, and run batches sequentially for operator diversity (Section 3). The traditional transition rule supports parallel manner well but fails to satisfy the requirement of serial manner. As shown in Figure 10(b), the stage executors always run in parallel even if only one buffer pair is used. Because $executor_i$ legitimizes bp_1 for $executor_{i-1}$ after it completes the execution, $executor_{i-1}$ is active with bp_1 , when $executor_{i+1}$ is in working state with bp_1 . In this case, $executor_{i-1}$ is able to run in parallel with $executor_{i+1}$.

We therefore design the transition diagram in Figure 11. Suppose $executor_i$ is in the active state with a legitimate buffer pair currently. **Once $executor_i$ pulls out a batch with buffer pair bp_j , it checks the states of bp_j instead of starting execution directly.** If bp_j is legitimate, $executor_i$ enters working state and starts the execution. Meanwhile, the input buffer and output buffer of bp_j enter in-reading and in-writing states respectively. **If bp_j is not legitimate, $executor_i$ enters checking state, waiting for bp_j to become legitimate.** When $executor_i$ completes a batch with bp_j , the input buffer of bp_j enters invalid state, and the output buffer of bp_j restores to its previous state. If $executor_i$ gets another legitimate buffer pair, it enters active state. Otherwise, it enters inactive state. The last stage executor always re-enters active state directly.

Note that, after $executor_i$ re-enters active state with any legitimate buffer pair, $executor_i$ updates the j -th input buffer of $executor_{i-1}$ to available state. It denotes that

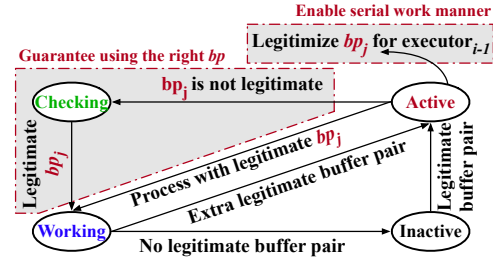


Figure 11: State transition diagram of $executor_i$ in DVABatch.

$executor_{i-1}$ can use bp_j now.

In short, we first add a *checking* state for the stage executor to guarantee using the right buffer pair. We also move the buffer pair legitimation for $executor_{i-1}$ after $executor_i$ enters active state. In this case, while DVABatch only configures one available buffer pair, all stage executors stay in inactive state until the current batch is executed by the last stage executor. Therefore, the serial work manner is supported.

6.3 Implementing the Transition Diagram

We implement the transition diagram of stage executors through CUDA synchronization APIs for the correctness guarantee. Each stage executor is bound to a CUDA *stream* for parallel execution, and each buffer pair is equipped with a CUDA *event* to enforce its legitimation. Upon finishing launching the CUDA functions with a buffer pair, the stage executor performs a *record* operation with the buffer pair's *event* on its *stream*. After that, if the corresponding buffer pair is legitimate, the other stage executor requires synchronization with the *event* on its own *stream* before using the buffer pair to avoid data hazards on GPUs. In order to deliver the best performance, the stage executor calls *cudaStreamWaitEvent* instead of explicit synchronizations on the host side.

7 Scheduling Policies of Serving Diversities

In this section, we present the way to deal with the serving diversities using DVABatch. First of all, we identify the existing diversities in a DNN model and divide the model into several stages. Then, at runtime, DVABatch is able to schedule the batches of the model appropriately.

7.1 Identifying Diversities and Slicing Models

For a DNN service, We identify the existing diversities and slice the model offline, by checking the input patterns and profiling the model with several different batch sizes (e.g., 1, 2, 4, 8, 16, 32, 64) using the tools provided by Nvidia [7, 8].

All the models are considered to have load diversities, as the load pattern is often determined by the end-users. The model that accepts inputs with different shapes (dynamic dimension

```

1 //stage executors run within a while loop
2 void Run():
3     Batch& inBatch = BatchQueue.Get();
4     CheckBuffer(inBatch); //Check buffer pair
5     Execute(inBatch);
6     Schedule(inBatch, outBatches);
7     for (auto& batch : outBatches):
8         nextBatchQueue.Push(batch)
9         getBuffer(); //get legitimate buffer pair
10        updatePrExecutor(); //update preceding executor
11 //call Schedule to perform meta operations
12 void Schedule(Batch& inBatch, vector<Batch>&
13 outBatches):
14     BatchTable.update(inBatch);
15     if userDefined1:
16         outBatches = BatchTable.New(inBatch);
17     else if userDefined2:
18         outBatches = BatchTable.Stretch(inBatch);
19     else if userDefined3:
20         outBatches = BatchTable.Split(inBatch);
21     else:
22         outBatches.Copy(inBatch);

```

Figure 12: Creating scheduling policies with meta operations.

except for batch size) has input diversity. During the profiling, we obtain the preferred batch size of each operator, as shown in Figure 5. When the operators have different preferred batch sizes, the model has operator diversity.

Once the diversities are identified, DVABatch slices the DNN models into stages. If a model is sliced into N_{st} stages, the operators are time-evenly assigned to the stages in the topological order for simplicity. It is non-trivial to theoretically identify the optimal N_{st} . If N_{st} is too small, the opportunity for batch scheduling is limited. Otherwise, if N_{st} is too large, the fine-grained stages incur heavy scheduling overhead. Moreover, unlike Pipedream [48], the model slicing in DVABatch can also be tight with diversities. E.g., DVABatch considers the operators’ preferred batch size and slices a model at specific operators for operator diversity. We currently determine the optimal N_{st} through profiling. It can be done in 10 minutes (8 tries) for emerging benchmarks on each type of GPU. It is worth noting that the model slicing does not conflict with the compilation techniques like kernel fusion [64]. We slice the model after it is already optimized by the DNN compilers.

7.2 Defining Policies with Meta Operations

Figure 12 shows the interface provided by DVABatch to define batch scheduling policies with the meta operations. Each stage executor runs in a while loop (Line 2-11) to execute the batches (Line 3, 6, 8 stated in Section 5, Line 4, 5, 9, 11 stated in Section 6). The stage executor calls `Schedule()` to schedule the batches at Line 5. Inside `Schedule()`, the stage executor updates the batch table, and performs meta operations if user-defined conditions are satisfied accordingly.

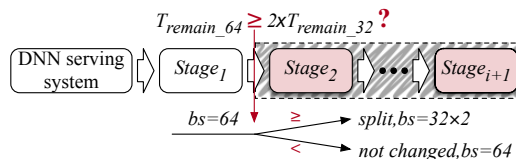


Figure 13: Scheduling according to the preferred batch size.

The policy is implemented outside user models, and does not require modification to the model. We then show the policies defined to handle the three diversities.

Policy I for input diversity. The input diversity requires running multiple small batches in parallel. While accepting a batch of queries from the upper-level serving systems, DVAScheduler clusters the queries according to their input sizes. The queries with similar input sizes are batched and padded to the same size for processing. DVABatch processes these batches in parallel for better utilizing the hardware parallelism. As the batches run in parallel, the scheduler prefers a smaller batch time window instead of generating the batches as large as possible. Practically, we set the batch time window to be the duration of the first stage with the largest allowed batch size bs_{max} . The number of active queries in the software pipeline does not exceed bs_{max} .

Policy II for operator diversity. Figure 13 shows the way DVABatch schedules the next stage when a stage completes. Assume the current batch size (denoted by bs) is 64 for $stage_1$ in the figure. The DVAScheduler compares the processing time of all the remaining stages with different batch sizes. Let T_{remain_i} represent the time needed when batch size is i . In Figure 13, if $T_{remain_64} \geq 2 \times T_{remain_32}$, the large batch is split into two batches with $bs = 32$. The two smaller batches run in the serial manner, as the hardware is already fully utilized with $bs = 32$. The duration of each stage with the different batch sizes is already profiled offline.

Policy III for load diversity. At load diversity, a latter batch should be able to join a previous batch, if it does not result in the QoS violation of the previous batch. In this case, DVABatch uses `stretch` operation to enlarge the batch at runtime. We set a time threshold T_{comp_wait} to eliminate the possible QoS violation. If a batch is already processed for T_{comp_wait} , no `stretch` operation is allowed on this batch.

Load diversity is widespread. For input diversity, as the new batches are allowed to enter the software pipeline independently, it already resolves the load diversity. For operator diversity, Policy III and Policy II work together due to the co-existence of `stretch` and `split`, as stated in Section 5.2.2.

8 Evaluation of DVABatch

In this section, we evaluate the performance of DVABatch in reducing the latencies of DNN services.

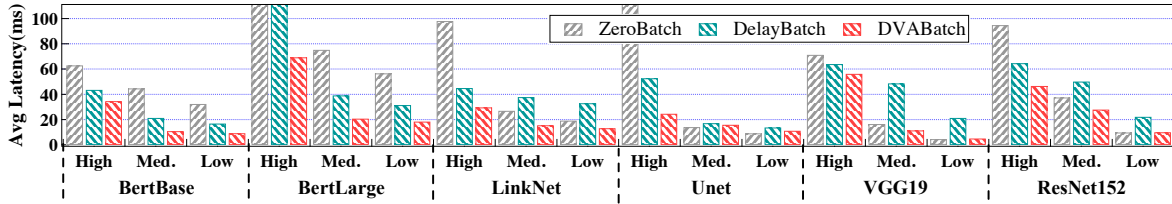


Figure 14: Average latencies of six DNN services at low, medium, high load with ZeroBatch, DelayBatch, and DVABatch.

Table 1: Evaluation specifications.

Hardware	CPU: Intel Xeon E5-2620, GPU: Nvidia Titan RTX
OS & Driver	Ubuntu: 18.04.6 (kernel 4.15.0); GPU Driver: 470.57
Software	CUDA: 11.4; TensorRT: 8.03; Triton 21.10
Benchmarks	Unet [56]; LinkNet [16]; BertBase; BertLarge [27]; VGG19 [57]; ResNet152 [37]
Dataset	GLUE [59]

8.1 Experiment Setup

We implement the prototype of DVABatch with 5k lines of C++ codes as a runtime backend for Triton [9], a DNN serving system from Nvidia. As the latency of a DNN model/operator varies with DNN frameworks [14, 19, 20, 39, 52] or compilers, we use TensorRT [13] to provide SOTA operator performance. DVABatch relies on Triton to batch the accepted queries. However, Triton sends the batched queries to DVABatch in an asynchronous fashion, and DVABatch enables the multi-entry multi-exit scheme for it. We also modify the model loading logic to load multiple stages (each stage is a sub-model) for a single DNN service.

Table 1 lists the setups of the experiments. We perform all the experiments on a machine that equips an Nvidia Turing Titan RTX (Titan RTX) GPU. We use six representative image processing and natural language processing DNN models as the benchmarks. All models experience load diversity. Besides, *BertBase* and *BertLarge* show input diversity, *LinkNet* and *Unet* show operator diversity.

We compare DVABatch with two batching policies: the default scheduling policy with batch time window $T_{window} = 0$ (ZeroBatch for short), and one with an optimized T_{window} (DelayBatch for short). The optimized T_{window} of DelayBatch is tuned for supporting the max peak throughput [2]. In all experiments, the maximum allowed batch size bs_{max} is 64 and the QoS target is 200ms to support a high load. Current production DNN serving systems (e.g., Triton [9], Clipper [23], TFServing [51]) all use the above batch time window and batch-size-based batching mechanism [22].

The load used for evaluation is generated using the method in MLperf [55], and the arrival time pattern satisfies the Poisson distribution [55]. We obtain the performance of the benchmarks at low, medium, and high loads. For a benchmark, we use 1/4, 3/5, 9/10 of its peak throughput as low load, medium

load, high load. The three load levels are obtained by feeding each benchmark with a stepping load [3] in Section 8.3. For BertBase and BertLarge, we use the workloads in GLUE [59] to simulate the sequence length distributions of real-world services. By default, the RTE workload of GLUE is used for evaluating BertBase and BertLarge. Other workloads are evaluated in Section 8.4.

8.2 Reducing Average Latency

Figure 14 shows the average latencies of all benchmarks with ZeroBatch, DelayBatch, and DVABatch at low, medium, high loads. DVABatch reduces the average latency of the benchmarks by 16.1%/39.0%/57.7% compared with ZeroBatch, 35.4%/47.3%/48.5% compared with DelayBatch on average at low, medium, and high loads, respectively. DVABatch reduces the average latency in all cases with the multi-entry multi-exit batching scheme. Meanwhile, DelayBatch shows lower average latency at high load compared with ZeroBatch, and ZeroBatch achieves lower latencies at low load. It is because DelayBatch has an optimized batch time window for peak throughput, and ZeroBatch does not introduce latency due to the batch time window at low load.

BertBase and BertLarge show both input diversity and load diversity. The latency reduction of DVABatch is comparatively high at all loads compared with the two baselines. This is because DVABatch perceives the input diversity, and splits the large batches into small batches to reduce the extra computation due to padding in all cases. DVABatch processes the small batches in the form of a software pipeline to accelerate the computation, which further reduces the latency.

LinkNet and Unet show both operator diversity and load diversity. At low load, DVABatch and ZeroBatch reduce latency due to the smaller batch window, compared with DelayBatch. At high load, DVABatch performs much better than ZeroBatch and DelayBatch. When the load is high, the batch received from the upper-level serving system has more queries. There is a higher opportunity that DVABatch can split a large batch into batches with the preferred batch size. Operators do not use a batch size larger than their preferred batch sizes. Some queries can exit the batching early with shorter latency.

VGG19 and ResNet152 only show load diversity. At low load, DVABatch achieves equivalent latency performance compared with ZeroBatch, and reduces the average latency

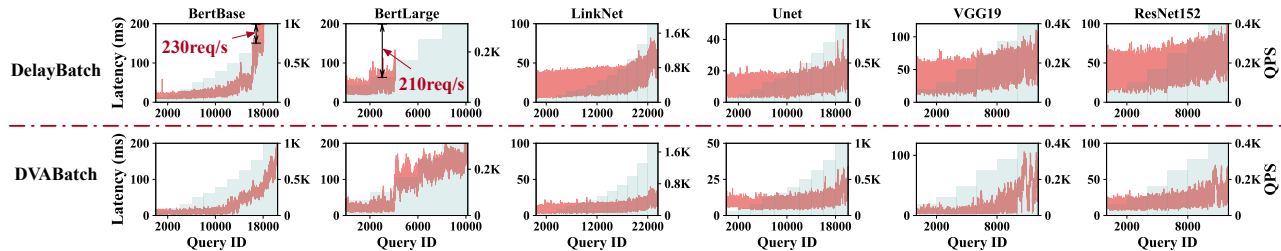


Figure 15: Latencies and peak throughput of DelayBatch and DVABatch fed with stepping load.

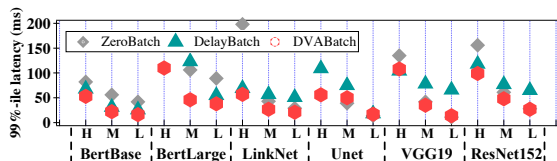


Figure 16: 99%-ile latencies of six DNN services at three loads with ZeroBatch, DelayBatch, and DVABatch.

compared with DelayBatch. This is reasonable because DVABatch and the baselines are all processing the queries with $bs = 1$ in this case. At high load, DVABatch and DelayBatch both perform better than ZeroBatch. The latency reduction benefits from the reasonable batching parameters. With DVABatch, early arrived queries are executed in advance, and latter queries are also considered to be merged with the former batch. Then, their response latencies are all reduced.

Comparison with Limited Solutions. While we do not compare DVABatch with BatchMaker [31] and LazyBatch [22] directly in this section, DVABatch outperforms them for the evaluated benchmarks. BatchMaker cannot handle input diversity for BertBase and BertLarge, as it is RNN-specific for input diversity and Bert-like models are not based on RNN cells. In this case, BatchMaker performs the same as the two baselines presented in Figure 14. LazyBatch only handles load diversity and performs per-operator scheduling. DVABatch degenerates to LazyBatch while evaluating VGG19 and ResNet152, if we slice the model into the granularity of operators and only enable *stretch* operation. However, experiments in Section 8.7 indicates this incurs severe performance degradation. Meanwhile, per-operator model slicing for LazyBatch cannot be implemented with TensorRT, as it is against the compilation techniques like kernel fusion. Compared with them, DVABatch achieves better performance for all evaluated benchmarks by promising a multi-entry multi-exit batch scheme with minimal runtime scheduling overhead.

Comparison of Tail Latency. Other than the average latency, Figure 16 shows the 99% latencies of all benchmarks. DVABatch reduces the 99%-ile latencies by 16.9%/27.4%/53.7% compared with ZeroBatch, and 45.2%/45.1%/29.2% compared with DelayBatch. In terms of tail latency, DelayBatch has better performance at high

load, ZeroBatch performs better than DelayBatch at low load. The multi-entry multi-exit design allows DVABatch to maintain consistent low tail latency at varying loads.

8.3 Robustness at Stepping Load

In this experiment, we evaluate the robustness of DVABatch in handling dynamic loads. Similar to prior work [3, 40], we use stepping load to obtain the peak load supported by DVABatch. We only compare DVABatch with DelayBatch in the following section, as ZeroBatch always shows poor performance at high load.

The stepping load is generated as follows. At first, the load is low (66 queries per second), and we gradually increase the load for every 2000 queries. After 30,000 queries, the load is increased to 4000 queries per second (QPS). We use the corresponding highest load under the constrain, that the latency is shorter than the QoS target $200ms$, as the peak throughput.

Figure 15 shows the latencies of the benchmarks at stepping loads. In each subfigure, the x -axis represents the query ID in the issuing order. The left y -axis represents the latency of each query, and the right y -axis represents the load.

As observed, all the benchmarks have lower latency with DVABatch than with DelayBatch in all cases. For BertBase and BertLarge, DVABatch improves the peak throughput, because it eliminates the computation wasted for the padded inputs. On average, DVABatch increases 46.81% peak throughput for BertBase, $1.37\times$ peak throughput for BertLarge. For operator diversity and load diversity, DVABatch has not impact on the computation. In that case, the peak throughput of DVABatch is limited by the hardware capacity. DVABatch maintains the same peak throughput as DelayBatch.

8.4 Impact of Input Distributions

The effectiveness of DVABatch for input diversities is affected by how different the inputs are. In this experiment, we show the performance of DVABatch when different workloads in GLUE [59] are used as the inputs of Bert and BertLarge. We use the same stepping load in Section 8.3.

Figure 17 shows the supported peak loads of BertBase and BertLarge with different workloads in GLUE. As observed,

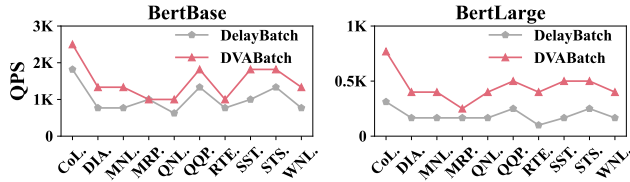


Figure 17: Peak load supported by DVABatch for BertBase/BertLarge with different workloads.

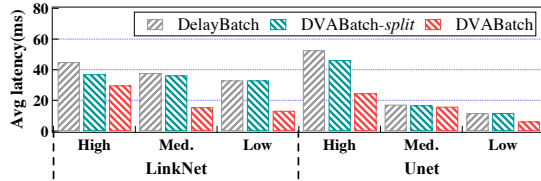


Figure 18: Comparison of *split* operation for LinkNet/Unet.

DVABatch improves the peak throughput by 46.8% for BertBase, $1.37\times$ for BertLarge compared with DelayBatch on average. DVABatch brings different throughput improvements for different datasets. The more imbalanced the sequence distribution is, the higher the workload’s input diversity is.

In general, DVABatch works better for workloads with higher input diversity, as DVABatch can eliminate more unnecessary padding. For instance, while the input diversity of SST-2 is higher than that of MRPC, the performance gap between DelayBatch and DVABatch for SST-2 workload is much larger than that for MRPC. The more imbalanced the sequence distribution is, the higher throughput improvement DVABatch achieves.

8.5 Effectiveness of *split* operation

In this experiment, we evaluate the effectiveness of *split* for load diversity. Figure 18 shows the average latencies of LinkNet and Unet with DVABatch, DVABatch-*split*, and DelayBatch. DVABatch-*split* is a variant of DVABatch that only *split* is enabled in DVABatch.

As shown, DVABatch-*split* reduces the latencies most at high load for LinkNet and Unet. Compared with DelayBatch, DVABatch-*split* reduces average latency by 15.0% at high load. This is because DVABatch-*split* rarely has the choice to split a batch at low and medium loads.

We can also find that DVABatch-*split* brings different improvements for the two DNN services. The difference originates from the operator diversity and load diversity. We look into the processing details and find that DVABatch-*split* generates larger batches for LinkNet than Unet. Half of the batches generated for LinkNet are with $bs > 50$ and Unet are with $bs > 30$. In this case, DVABatch-*split* has more chances to identify the preferred batch size for LinkNet than Unet.

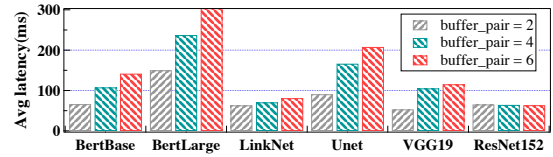


Figure 19: The average latencies of DVABatch with different number of buffer pairs under peak load.

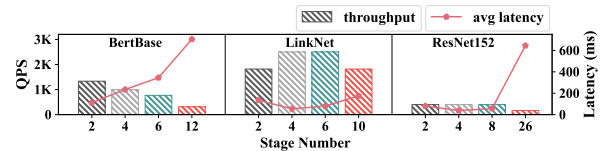


Figure 20: The average latencies and peak throughput of the benchmarks when sliced into different numbers of stages.

8.6 Impacts of the Number of Buffer Pairs

This experiment evaluates the impact of the number of buffer pairs used in DVABatch. Figure 19 presents the average latencies of DVABatch with different numbers of buffer pairs at the peak load. As shown, the average latencies are always the lowest with two buffer pairs for all benchmarks.

Therefore, two buffer pairs are already enough to preserve the execution validity and enable the work manner switch. More buffer pairs degrade the performance. Each buffer pair uses a set of CUDA [50] synchronization data structures to guarantee scheduling correctness. When two buffer pairs are used, DVABatch only needs to switch between them. However, managing many buffer pairs requires extra FIFO queues to transmit these data structures. Too many buffer pairs incur a high scheduling overhead.

8.7 Impacts of the Stage Numbers

In this experiment, we investigate the number of stages on the performance of DVABatch. We use BertBase, LinkNet, and ResNet152 as the representative benchmarks with the three types of diversities, respectively, due to the limited space.

Figure 20 shows the throughputs and average latencies of the benchmarks with different stage numbers. In the figure, the left y-axis represents the peak throughput and the right y-axis represents the corresponding average latencies.

As observed, the best stage number is 2 for BertBase, 4 for LinkNet and ResNet152 in terms of latency and throughput, respectively. For BertBase, the accepted batch is *split* into two batches in most cases. DVABatch avoids generating too many small batches to reduce scheduling overhead. Therefore, 2 stages are enough for BertBase. LinkNet and ResNet152 require more stages to enable *stretch* and *split*. If the number of stages is too large (e.g., 20), managing queues between stages incur a high overhead for all benchmarks.

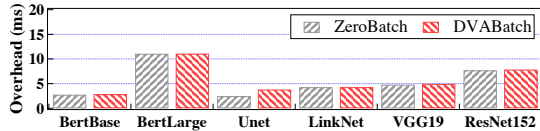


Figure 21: Close-loop latencies of ZeroBatch and DVABatch.

8.8 Scheduling Overhead

As mentioned in Section 7.1, the profiling for model slicing needs to be done for a single time on each type of target GPU, and completed in 10 minutes. To measure the runtime overhead introduced by DVABatch, we run the six benchmarks in a close loop, and compare the end-to-end latencies of the queries with ZeroBatch. In this experiment, we set the batch time window of DVABatch to 0. Figure 21 shows the experimental results.

As observed, the average latency overhead is 0.29ms, and DVABatch achieves almost the same latency compared with ZeroBatch. DVABatch has a low overhead because it does not interrupt the execution of stage executors at the extremely low load, and the two executors overlap the overhead of CPU-GPU synchronization for each stage. As depicted in Figure 12, *Line5* is usually an asynchronous operation on GPU and the scheduling operation in *Line4* is overlapped by the execution. Moreover, the model slicing in DVABatch does not invalidate the optimization of DNN compilers.

DVABatch needs extra global memory (buffer pairs) to avoid read-write hazards while maintaining the software pipeline, which takes 200 MB of space on average.

9 Discussion

9.1 Implication for Future DNN Inference

Omnipresent Diversity. Readers can find that all the mentioned diversities are caused by dynamic attributes (dynamic input, operator, load). Existing DNNs may have a dynamic architecture in depth, width, and routing [32,33,45,53,58,60]. As more dynamic attributes emerge, diversity spreads across new DNNs.

Intra-model Scheduling. DVABatch performs fine-grained scheduling within the DNN models. As DNNs grow larger and show more diversity, the execution of DNNs cannot be treated as a single function call. Large models [15,54] like Bert are being deployed on multiple machines. MoE [42] models activate different paths for different inputs. Intra-model scheduling is a trend for future DNN inference.

9.2 Flexibility

DVABatch is flexible to other diversities. For instance., in the services with early-exiting and layer-skip[42,53,55] models,

the inference returns at early stages or skips some stages when the intermediate results satisfy a predefined threshold. Users can modify the user-defined condition in DVABatch to check the intermediate results during batch inference. If some queries meet the predefined threshold, users then utilize the *split* operation for them to exit the ongoing batch without executing the rest layers. Then, the layer skip mechanism is implemented by the holistic design of multi-entry and multi-exit in DVABatch.

9.3 Limitations

DVABatch targets on efficient batch processing of models with diversities. It performs the same as the traditional batch policy for the models without any diversity. The CV models commonly crop the images to the same size before processing. Then input diversity does not exist in these models. Because the same blocks share the favored batch size, models with repetitive blocks like ResNet-50 and Bert do not show operator diversity. Load diversity also vanishes when the queries arrive with a uniform load. In these cases, DVABatch has no opportunity to achieve a better batch scheme. But as stated in Section 9.1, more and more model are showing diversities due to dynamic attributes. As long as a more efficient batch scheme is available for the diversities, DVABatch takes effect through its holistic design, even on platforms like CPUs [61].

10 Conclusion

In this work, we utilize the multi-entry multi-exit scheme to resolve the long latency problem due to serving diversity in existing DNN serving systems. We dig out the root inefficiency of the existing batching policy when facing serving diversities. Therefore, we propose DVABatch runtime batching system. Firstly, DVABatch divides the DNN models into stages and abstracts three meta operations to support the multi-entry multi-exit scheme. Secondly, DVABatch introduces a state transition diagram to manage the execution of stages. And then, DVABatch conducts diversity-aware batch scheduling with the meta operations for the incoming batch of queries. Overall, DVABatch achieves 46.4% average latency reduction and up to $2.12\times$ throughput improvement for involved diversities, compared with state-of-art solutions.

Acknowledgment

This work is partially sponsored by the National Natural Science Foundation of China (62022057, 61832006, 61872240, 62172375), Shanghai international science and technology collaboration project (No.21510713600), and Open Research Projects of Zhejiang Lab (No. 2021KE0AB02). Quan Chen, and Deze Zeng are the corresponding authors.

References

- [1] <https://aws.amazon.com/machine-learning/>.
- [2] Delayed batching in triton. https://github.com/triton-inference-server/server/blob/v2.14.0/docs/model_configuration.md#delayed-batching.
- [3] Edit load patterns to model virtual user activities. <https://docs.microsoft.com/en-us/visualstudio/test/edit-load-patterns-to-model-virtual-user-activities?view=vs-2022>.
- [4] Effective transformer. https://github.com/bytedance/effective_transformer.
- [5] Google translate. <https://translate.google.com>.
- [6] Microsoft xiaoice. <http://www.msxiaobing.com/>.
- [7] Nvidia nsight compute. <https://docs.nvidia.com/nsight-compute/NsightCompute/index.html>.
- [8] Nvidia nsight system. <https://developer.nvidia.com/nsight-systems>.
- [9] Nvidia triton inference server. <https://github.com/NVIDIA/triton-inference-server>.
- [10] Nvidia turing gpu architecture whitepaper. <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf>.
- [11] Nvidia v100 tensor core gpu. <https://www.nvidia.com/en-us/data-center/v100/>.
- [12] Siri. <https://www.apple.com/siri/>.
- [13] Tensorrt. <https://developer.nvidia.com/tensorrt>, 2021.
- [14] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.
- [15] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Nee-lakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.
- [16] Abhishek Chaurasia and Eugenio Culurciello. Linknet: Exploiting encoder representations for efficient semantic segmentation. In *2017 IEEE Visual Communications and Image Processing (VCIP)*, pages 1–4. IEEE, 2017.
- [17] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the 22th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 17–32, 2017.
- [18] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ACM SIGPLAN Notices*, 51(4):681–696, 2016.
- [19] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.
- [20] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.
- [21] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE, 2014.
- [22] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. Lazybatching: An sla-aware batching system for cloud machine learning inference. *arXiv preprint arXiv:2010.13103*, 2020.
- [23] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.
- [24] Weihao Cui, Quan Chen, Han Zhao, Mengze Wei, Xiaoxin Tang, and Minyi Guo. E²bird: Enhanced elastic batch for improving responsiveness and throughput of deep learning services. *IEEE Trans. Parallel Distributed Syst.*, 32(6):1307–1321, 2021.

- [25] Weihao Cui, Mengze Wei, Quan Chen, Xiaoxin Tang, Jingwen Leng, Li Li, and Mingyi Guo. Ebird: Elastic batch for improving responsiveness and throughput of deep learning services. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 497–505. IEEE, 2019.
- [26] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Enable simultaneous DNN services based on deterministic operator overlap and precise latency prediction. In *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, 2021*, pages 15:1–15:15. ACM, 2021.
- [27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [28] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbo Transformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.
- [29] Pratik Fegade, Tianqi Chen, Phillip B Gibbons, and Todd C Mowry. The cora tensor compiler: Compilation for ragged tensors with minimal padding. *arXiv preprint arXiv:2110.10221*, 2021.
- [30] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.
- [31] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.
- [32] Yue Guan, Zhengyi Li, Jingwen Leng, Zhouhan Lin, and Minyi Guo. Transkimmer: Transformer learns to layer-wise skim. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022*, pages 7275–7286. Association for Computational Linguistics, 2022.
- [33] Cong Guo, Yuxian Qiu, Jingwen Leng, Xiaotian Gao, Chen Zhang, Yunxin Liu, Fan Yang, Yuhao Zhu, and Minyi Guo. SQuant: On-the-fly data-free quantization via diagonal hessian approximation. In *International Conference on Learning Representations*, 2022.
- [34] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 982–995. IEEE, 2020.
- [35] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40. IEEE, 2015.
- [36] Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, et al. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–238, 2015.
- [37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [38] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [39] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.
- [40] Zhen Ming Jiang and Ahmed E Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118, 2015.
- [41] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.
- [42] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

- [43] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670, 2014.
- [44] Xiqing Li, Guangyan Zhang, H Howie Huang, Zhufan Wang, and Weimin Zheng. Performance analysis of gpu-based convolutional neural networks. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 67–76. IEEE, 2016.
- [45] Lanlan Liu and Jia Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
- [46] Zihan Liu, Jingwen Leng, Zihui Zhang, Quan Chen, Chao Li, and Minyi Guo. VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022*, pages 388–401. ACM, 2022.
- [47] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.
- [48] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019*, pages 1–15. ACM, 2019.
- [49] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.
- [50] CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.
- [51] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.
- [52] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.
- [53] Yuxian Qiu, Jingwen Leng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. Adversarial defense through network profiling based path extraction. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019*, pages 4777–4786. Computer Vision Foundation / IEEE, 2019.
- [54] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.
- [55] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459. IEEE, 2020.
- [56] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.
- [57] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [58] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469. IEEE, 2016.
- [59] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.
- [60] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 409–424, 2018.
- [61] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. Deepcpu: Serving rnn-based deep learning models 10x faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 951–965, 2018.
- [62] Wei Zhang, Quan Chen, Ningxin Zheng, Weihao Cui, Kaihua Fu, and Minyi Guo. Toward qos-awareness and

improved utilization of spatial multitasking gpus. *IEEE Trans. Computers*, 71(4):866–879, 2022.

- [63] Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Edward Mawhirter, Bo Wu, Chao Li, and Minyi Guo. Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters. In *Proceedings of the ACM International Conference on Supercomputing, ICS 2019*, pages 58–68. ACM, 2019.
- [64] Han Zhao, Weihao Cui, Quan Chen, Youtao Zhang, Yanchao Lu, Chao Li, Jingwen Leng, and Minyi Guo. Tacker: Tensor-cuda core kernel fusion for improving the GPU utilization while ensuring qos. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022*, pages 800–813. IEEE, 2022.
- [65] Han Zhao, Weihao Cui, Quan Chen, Jieru Zhao, Jingwen Leng, and Minyi Guo. Exploiting intra-sm parallelism in gpus via persistent and elastic blocks. In *39th IEEE International Conference on Computer Design, ICCD 2021*, pages 290–298. IEEE, 2021.

A Artifact Appendix

Abstract

This artifact provides a prototype of DVABatch implemented as a runtime backend for Triton Inference Server

Scope

This artifact is licensed with Apache 2.0

Hosting

The code is available on Github <https://github.com/sjtu-epcc/DVABatch.git>.

Requirements

Hardware Requirements.

1. CPU: Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
2. Memroy: 252G
3. NVIDIA TitanRTX

Software Requirements.

1. Ubuntu 18.04.6 (Kernel 4.15.0)
2. GPU Driver: 460.39
3. CUDA 11.3
4. CUDNN 8.2
5. TensorRT 8.0.3.4
6. RapidJSON
7. Cmake 3.17

Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing

Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon and Jaehyuk Huh
School of Computing, KAIST
{sbchoi, myshlee417, yjkim, jspark}@casys.kaist.ac.kr; {yjkwon, jhhuh}@kaist.ac.kr

Abstract

As machine learning (ML) techniques are applied to a widening range of applications, high throughput ML inference serving has become critical for online services. Such ML inference servers with multiple GPUs pose new challenges in the scheduler design. First, they must provide a bounded latency for each request to support a consistent service-level objective (SLO). Second, they must be able to serve multiple heterogeneous ML models in a system, as cloud-based consolidation improves system utilization. To address the two requirements of ML inference servers, this paper proposes a new inference scheduling framework for multi-model ML inference servers. The paper shows that with SLO constraints, GPUs with growing parallelism are not fully utilized for ML inference tasks. To maximize the resource efficiency of GPUs, a key mechanism proposed in this paper is to exploit hardware support for spatial partitioning of GPU resources. With spatio-temporal sharing, a new abstraction layer of GPU resources is created with configurable GPU resources. The scheduler assigns requests to virtual GPUs, called *gpulets*, with the most effective amount of resources. The scheduler explores the three-dimensional search space with different batch sizes, temporal sharing, and spatial sharing efficiently. To minimize the cost for cloud-based inference servers, the framework auto-scales the required number of GPUs for a given workload. To consider the potential interference overheads when two ML tasks are running concurrently by spatially sharing a GPU, the scheduling decision is made with an interference prediction model. Our prototype implementation proves that the proposed spatio-temporal scheduling enhances throughput by 61.7% on average compared to the prior temporal scheduler, while satisfying SLOs.

1 Introduction

The wide adoption of machine learning (ML) techniques poses a new challenge in server system designs. Traditional server systems have been optimized for CPU-based computa-

tion for many decades. However, the regular and ample parallelism in widely-used deep learning algorithms can exploit abundant parallel execution units in GPUs. Powerful GPUs have been facilitating the training computation of deep learning models, and the inference computation is also moving to the GPU-based servers due to the increasing computational requirements of evolving deep learning models with deeper layers [11, 38, 45].

However, the GPU-based inference servers must address different challenges from the batch-oriented processing in ML training servers. First, inference queries must be served within a bounded time to satisfy service-level objectives (SLOs). Therefore, not only the overall throughput is important, but bounded response latencies for processing inference queries are also critical to maintain consistent service quality [15, 38, 45]. Second, to improve the utilization of server resources, many heterogeneous models are served by consolidated cloud-based systems. As even a single service can include multiple heterogeneous ML models [38], multiple models with different purposes coexist in a system. The heterogeneity of ML models raises scheduling challenges to map concurrent requests of heterogeneous models to multiple GPUs. Incoming queries for different ML models with their own computational requirements, must be properly routed to the GPUs to meet the SLO, while improving the overall throughput. In addition, the number of required GPU nodes must be dynamically adjusted to reduce the cost of serving inferences for cloud-based systems.

While the demands for GPU-based ML inferences have been growing, the computational capability of GPUs with many parallel execution units has been improving precipitously. Such ample parallel execution units combined with increasing GPU memory capacity allow multiple ML models to be served by a single GPU. In a prior study [38], more than one model can be mapped to a GPU, as long as the GPU can provide the execution throughput to satisfy the required SLO. However, unlike CPUs which allow fine-grained time sharing with efficient preemption, GPUs perform only coarse-grained kernel-granularity context switches. Such coarse-grained time

sharing incurs inefficient utilization of enormous computational capability of GPUs, as a single batch of an ML inference may not fill the entire GPU execution units.

However, the recent advancement of GPU architecture opens a new opportunity to better utilize the abundant execution resources of GPUs. Recent GPUs support an efficient spatial partitioning of GPUs resources (called MPS mechanism in NVIDIA GPUs [12]). The partitioning mechanism supports the computational resources of a GPU to be divided to run different contexts simultaneously. Such a unique spatial partitioning mechanism can augment the limited coarse-grained time sharing mechanism, as the GPU resource can be spatially partitioned to serve multiple ML tasks concurrently. This unique spatial and coarse-grained temporal resource allocation in GPUs calls for a novel abstraction to represent partitioned GPUs and a new scheduling framework targeting high throughput ML servers under SLO constraints.

To address the emerging challenges of ML scheduling in partitionable GPUs, this paper proposes a new abstraction for GPUs called *gpulet*, which can create multiple virtual GPUs out of a single physical GPU with spatial and temporal partitioning. The new abstraction can avoid the inefficiency of coarse-grained time sharing by creating and assigning the most efficient GPU share for a given ML model. Such a new abstraction of GPU resources allows latencies of ML execution to be predictable even when multiple models are concurrently running in a GPU, while achieving improved GPU utilization.

Based on the *gpulet* concept, we propose an ML inference framework prototyped with the PyTorch interface. It can serve concurrent heterogeneous ML models in multi-GPU environments with auto-scaling support. Figure 1 illustrates the extended search space of our scheduling mechanism. Our framework aims to find a global optimum by considering both temporal and spatial scheduling for enhanced performance. The search space for scheduling becomes three-dimensional with spatial and temporal shares of GPU resources in addition to batch size adjustment, unlike the prior work with two-dimensional searches [17, 38]. In the experimental results for SLO-preserved throughput presented by Figure 13, time scheduling and spatial scheduling yield on average 1,023 and 1,076 requests-per-second (RPS), respectively. The spatio-temporal scheduling significantly improves the SLO-preserved throughput to 1,584 RPS.

For each ML model, its computational characteristics are measured and registered to the framework. Based on the profiled information of each ML model, the scheduler routes requests to where the throughput would be maximized while satisfying the SLO constraints. One necessary mechanism for spatial and temporal partitioning of GPU shares is to identify the potential performance overheads when two models are concurrently running on a GPU. Our scheduling framework incorporates the interference estimation mechanism to consider the effect of concurrent execution.

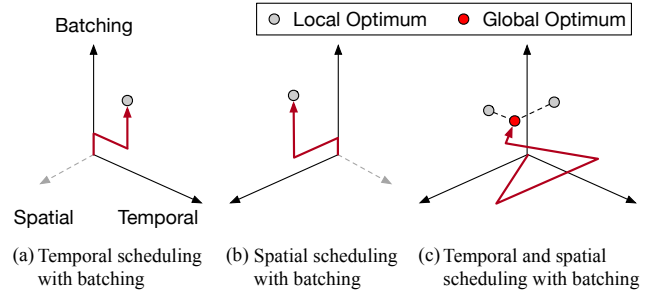


Figure 1: Multi-dimensional search space for providing globally optimal performance.

We evaluated the proposed ML inference framework on server systems with four and eight GPUs. The evaluation with four GPUs shows that the proposed scheduling technique with *gpulets* can improve the throughput with SLO constraints for seven ML inference scenarios by 61.7%, compared to the one without partitioning GPU resources.

This study explores a new resource provisioning space of GPUs for machine learning inference serving. The contributions of this paper are as follows:

- It proposes a new GPU abstraction named *gpulet*, to support virtual GPUs with partitions of resources out of physical GPUs. It allows heterogeneous ML models to be mapped to multiple *gpulets* in the most cost-effective way.
- It proposes a scheduling framework for *gpulets*, which searches the most cost-effective schedule by multi-dimensional search considering batch sizes, temporal sharing, and spatial sharing. It adjusts the number of required GPUs for a given set of heterogeneous models, supporting auto-scaling.
- For accurate performance prediction, the scheduling framework considers the effect of interference among *gpulets* for concurrent ML inference execution on partitions of a single GPU.

The rest of the paper is organized as follows. Section 2 describes the background of ML computation on GPUs and prior scheduling techniques. Section 3 presents the motivational analysis of heterogeneous ML tasks on multiple GPUs. Section 4 proposes our design to efficiently utilize GPU resources for heterogeneous ML tasks, and Section 5 presents experimental results. Section 6 presents related work, and Section 7 concludes the paper.

2 Background

2.1 Batch-Aware ML Inference Serving

As high throughput ML inference serving has become widely required for online services, an increasing number of service vendors are adopting GPUs [9, 13, 15, 24, 25, 32, 38, 41, 42, 45, 46] or even hardware accelerators such as TPUs [3,

7, 18, 19]. While GPU-based systems offer low latency for ML inference, obtaining high utilization is a challenging task, unlike ML training. The key difference between training and inference in terms of GPU utilization is the suitability for *batching*. For training, since the input data is ready, the system can batch a large number of input data, which allows GPUs to effectively leverage the massive parallelism. On the contrary, the ML inference server underutilizes GPUs as it is an on-demand system where inference tasks can be assigned to the compute engines once the requests arrive.

One scheduling option is to wait until the desired number of inference requests to be accumulated and then to initiate the execution for the large batch. However, the applications cannot indefinitely wait to collect a batch, due to the service-level objective (SLO) requirements. Prior work [15, 36, 38, 45] have adopted *adaptive batching*, where a batch size is decided adaptively with estimated times to build and execute a batch of the selected size. By using the profiled latencies and observed incoming rates, the effective time for a batch is estimated, and adaptive batching chooses the maximum batch size that does not violate the SLO.

2.2 Temporal Scheduling for ML on GPUs

Temporal scheduling allows sharing of a GPU where each inference takes up the entire GPU resource one at a time with time sharing. With multiple models with different execution characteristics and SLO requirements, guaranteeing SLO is challenging for the temporal scheduling of the heterogeneous models. The ML inference scheduling problem on GPU-based multi-tenant serving systems resembles the traditional bin packing problem. The capacity constraints of bins are the available resource on the GPUs, and an item weight is the necessary GPU resource to handle a given inference request.

An inspiring prior work, Nexus [38], has tackled this problem and proposed a novel variant of bin packing algorithm, namely *squishy bin packing* (SBP). The term, *squishy*, is originated from the property that the required resource for processing a task (i.e., item) and its latency vary as the batch size changes. The SBP algorithm takes a set of models as input, each of which comes with a given request rate. It assigns the inferences tasks across GPUs with a selected batch size, and may map multiple models to a single GPU with time sharing, if one task does not fully utilize a GPU.

Figure 2 illustrates an example of how the SBP algorithm is applied. In this scenario, the server handles two models, A and B, by building and executing the per-model batches simultaneously. The SLO violation occurs when the summation of batch building time and batch execution time exceeds the SLOs. Therefore, the SBP algorithm heuristically finds a maximum possible duration for batch building, called *duty cycle*, and the corresponding batch sizes in such a way that all the consolidated models would not violate the SLOs. The SBP algorithm repeats the duty cycles in a pipelined fashion

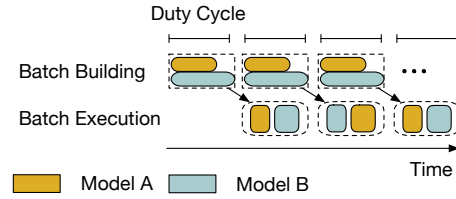


Figure 2: Round-based execution of SBP for two models consolidated on a GPU. *Duty cycle* is the interval for a round.

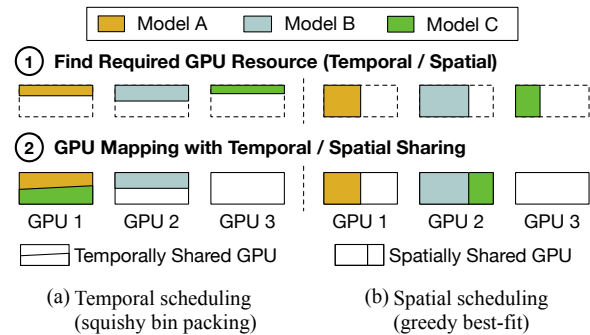


Figure 3: Temporal scheduling (SBP) vs. spatial scheduling (greedy best-fit partitioning).

until there is a significant change in the request rates, which would require rescheduling.

Baseline temporal scheduling algorithm: We use SBP as our baseline temporal sharing algorithm. Figure 3 (a) presents temporal sharing with batch adjustment by the SBP algorithm. The algorithm finds individual duty cycles and batch sizes for ML models with given request rates, and maps them to a minimum number of GPUs with temporal sharing. To map ML models to GPUs, it compares all eligible pairwise combinations. If a pair is eligible for temporal sharing, the pair will share one GPU and the batch size is further adjusted to ensure the SLO when both tasks are interleaved. The process continues until no more pairs can be temporally shared.

2.3 Spatial Sharing on GPU

Spatial sharing is a resource partitioning technique that splits a GPU resource into multiple pieces, as recent server-scale GPUs offer hardware-supported spatial sharing features to users. While temporal scheduling may potentially cause a GPU underutilization problem when the batch size is not sufficiently large to leverage all resources on a GPU, spatial sharing improves GPU utilization allowing high throughput without SLO violation.

With these resource partitioning features, the users can split the given resource of a GPU into a set of GPU *partitions*, each of which is assigned to a fraction of GPU resource¹. Note

¹In this paper, we only use the computation resource partitioning technique since we have at our disposal 2080 Ti GPUs, the microarchitecture of

that GPU *partitioning* is available on both NVIDIA MPS and Multi-Instance GPU (MIG), which has been featured since Ampere architecture GPUs. MIG provides physical partitioning with multi-GPU abstraction, while MPS provides logical execution resource partitioning by percentage. With physical partitioning, MIG allows partitions of memory capacity, memory bandwidth, and caches to be dedicated to each instance, in addition to execution cores.

A prior work, GSLICE [17], leverages GPU partitioning to increase throughput and utilization of GPUs. GSLICE employs a self-tuning algorithm for adjusting the amount of GPU resources per partition based on performance feedback. After adjusting the amount of resources, the batch size is heuristically decided by the SLO for the given task. However, the solution provided in GSLICE uses only spatial sharing and is limited to a single GPU.

Baseline spatial scheduling algorithm: As a baseline spatial sharing algorithm for our multi GPU framework, we use the *greedy best-fit* algorithm. Greedy best-fit algorithm chooses the minimum required partition size for each model to handle a given request rate with SLO constraint. It allocates the partitions of multiple ML models to GPUs through best-fit searching. Figure 3 (b) presents the spatial scheduling used by the greedy best-fit algorithm. Unlike the SBP or greedy-best fit algorithm, our scheduling scheme aims to simultaneously employ both temporal and spatial scheduling to maximize utilization and minimize the number of required GPUs.

3 Motivation

3.1 Optimal Batch Size and Partition

To understand the performance implications of batching and GPU partitioning, we perform an experimental study, using four ML models: GoogLeNet, ResNet50, SSD-MobileNet-V1, and VGG-16. The detailed descriptions for the ML models and GPU server specifications are provided in Section 5.1.

Figure 4 shows the batch inference latency results as the batch size increases from 1 to 32. For each batch size, we sweep through the increasing fractions of GPU resources, ranging from 20% to 100% to observe how the batch size and computing resource utilization are correlated. When the batch size is large, the latency significantly drops as more resource is added. The large slope of the curves implies that the inference execution for the particular batch size can use the additional resource effectively to reduce the latency. On the contrary, with a small batch, the latency is not largely affected by the amount of GPU resources, which implies that the GPU resource becomes underutilized when larger fractions are assigned. Hence, both batch size and amount of GPU resource must be considered as a joint factor when making cost-effective scheduling decisions.

which is Turing, an older generation than Ampere that offers the memory bandwidth isolation feature.

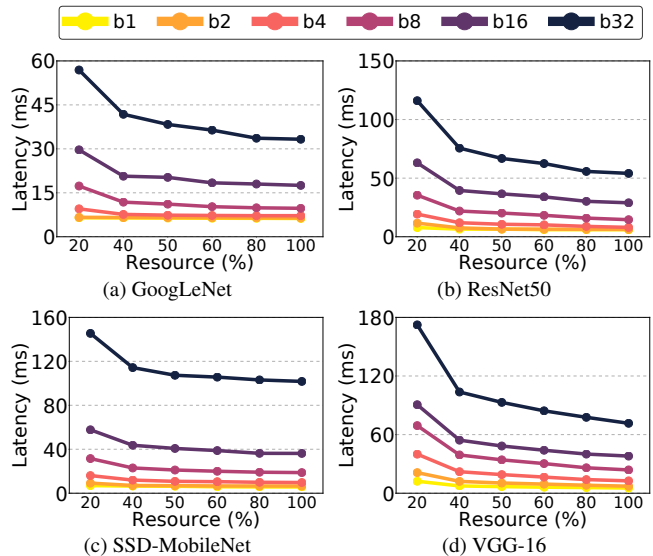


Figure 4: Batch inference latency as the fraction of computing resource assigned to the model inference changes from 20% to 100%, for the four ML models. Each curve represents a different batch size, and bn is a batch size of n .

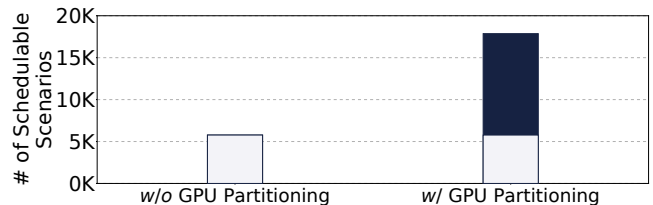


Figure 5: Number of schedulable scenarios when the SBP algorithm performs the scheduling *without* (left) and *with* (right) a fixed 1:1 GPU partitioning scheme.

3.2 Schedulability and GPU Partitioning

For a given set of SLO latencies for ML models, if incoming request rates are beyond the level that an inference server can cope with, the SLO will not be met as requests cannot be served on time. We define *schedulability* as the capability of a scheduling algorithm for serving the given request rates while not violating the SLO. A scheduler can improve schedulability by having better GPU utilization and in turn, having higher throughput with SLO satisfaction. To investigate the potential of GPU partitioning on the schedulability improvement, we evaluate a large number of possible multi-model inference serving scenarios. A *schedulable* scenario is the one in which the scheduler can successfully make a decision for the given rate while preserving SLO.

For the evaluation, we use nine models, and each corresponding SLO latency is listed in Table 3. For each scenario, models have one of the following request rates: 0, 100, and 200 requests per second (req/s). Since the zero req/s is in-

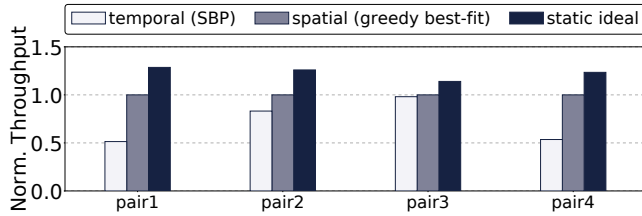


Figure 6: Comparison of SLO preserved throughput for temporal (SBP), spatial (greedy best-fit), and static ideal scheduling, normalized to spatial scheduling.

cluded in the set of possible request rates, we assume that a subset of the nine models may not be served at all. Excluding the scenario where all the models have zero request rate, we obtained total 19,682 ($= 3^9 - 1$) scenarios for the experiment.

Figure 5 reports the number of schedulable scenarios when we use the two different scheduling algorithms: 1) the default SBP algorithm without GPU partitioning support, and 2) the SBP algorithm with GPU partitioning. In this motivational result, a GPU is split into two partitions with the same resource in each partition, although our scheduler later will use a wider range of partitioning of GPUs. With the fixed 1:1 GPU partitioning, schedulable scenarios increased significantly from 5,772 with SBP to 19,682 by SBP with two partitions. Even though the GPU is partitioned with a fixed 1:1 ratio, the results show that GPU partitioning is capable of putting wasted GPU compute power to use, enabling higher resource utilization.

3.3 Performance of Effective Partitioning

To demonstrate how a cost-effective partitioning scheme affects performance, we compare SLO preserved throughput of three scheduling schemes: temporal (SBP), spatial (greedy best-fit), and ideal scheduling. The SLO preserved throughput is the maximum throughput sustainable by a system while supporting SLOs for all models running in the system. Figure 6 presents the normalized SLO preserved throughput with the three scheduling schemes. We use two GPUs for this experiment, and a pair of ML models are scheduled. The models are selected from Table 3. The pair used for the experiment are (1) *ssd/be*, (2) *res/vgg*, (3) *goo/mob*, and (4) *nas/den*.

The first scheme, temporal scheduling, does not partition GPUs, but schedules tasks in a time-sharing manner with the SBP algorithm. The second schemes partitions GPUs by our baseline spatial scheduling algorithm (greedy best-fit) introduced in Section 2.3. The last scheme, static ideal exhaustively searches all possible GPU partitioning ratios among (2:8), (4:6), and (5:5) for two GPUs. For each pair of tasks, it uses a GPU partitioning option which yields the highest performance. For these selected sets of ML models, the spatial scheduling (greedy best-fit) outperforms the temporal scheduling (SBP) by 51% on average, proving the performance benefits of spatial sharing. The static ideal scheduling shows 23%

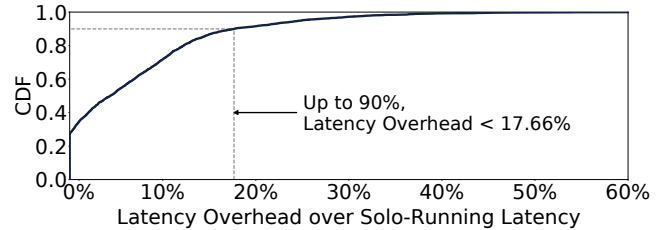


Figure 7: Cumulative distribution of latency overhead when pairs of inference executions are consolidated on a GPU.

improvements on average, compared to greedy best-fit. This experiment shows only limited example scenarios, but the results imply that partitioning can potentially improve the GPU utilization for certain scenarios, and better scheduling can further improve spatial scheduling.

3.4 Interference in Consolidated Executions

Cost-effective GPU partitioning allows enhancing the schedulability of SBP significantly. However, one important downside is the *performance interference* caused by multiple inference executions concurrently running on a GPU. One common cause of such performance interference is the bandwidth sharing for the external memory, but other contentions on on-chip resources may engender performance degradation too. To identify the interference effects, we perform an additional preliminary experimental study using a set of scenarios with ML models running together on a GPU. The model pairs were chosen among five models from Table 3: GoogLeNet, LeNet, ResNet50, SSD-Mobilenet, and VGG-16 (i.e., $C(5, 2) = 10$). Each pair runs with five different batch sizes (i.e., 2, 4, 8, 16, 32) creating 250 unique pairs in total. We also partition a GPU into two partitions using five different ratios: (2:8), (4:6), (5:5), (6:4), and (8:2). Then, we map the ML model pairs to the different partition pairs to observe the interference effects in various settings.

Figure 7 presents the cumulative distribution function (CDF) of latency overheads due to the consolidated inference executions, in comparison with the case where the models are run independently. As noted in the figure, for 90% of the scenarios, the interference-induced overhead is modest (lower than 18%). However, the CDF reports the long tail, suggesting that the interference effect could be severe in certain circumstances. In such cases, the interference may cause incorrect scheduling decisions, and the interfered task would experience latencies that are largely off from the expected range. Motivated by the insight, we devise an interference model and leverage it to make the scheduling decisions more robust, which reduces SLO violation rates.

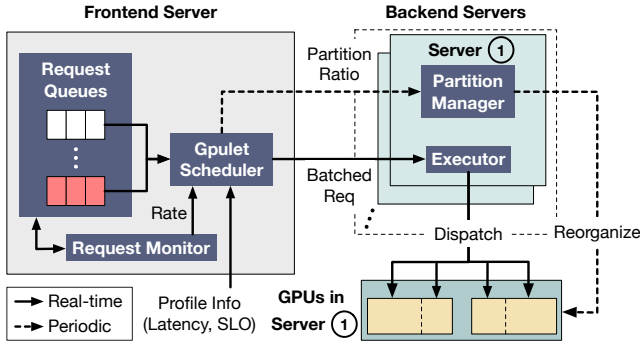


Figure 8: Overview of the scheduling framework with gplets.

4 Design

4.1 Overview

The goal of this study is to devise a scheduling framework for multi-model ML inference serving, which aims to assign incoming inference requests to the minimal number of GPUs while maximizing their utilization. The scheduling of ML inference workloads with SLO requirements must consider three aspects: batching, temporal scheduling, and spatial scheduling. Unlike the prior approaches which consider a subset of the three dimensions [17, 38], we propose a scheduler that fully explores all three dimensions to find the most effective point for scheduling.

In this study, using the spatial sharing capability of GPUs, we introduce an abstraction of GPU partition, called *gpulets*. Multiple gpulets can utilize a GPU by both temporal and spatial sharing. For each trained ML model, a minimal performance profile is collected offline. Based on the profiles of models, the scheduler distributes tasks to gpulets across multiple physical GPUs. Our scheduling framework minimizes the number of required physical GPUs while satisfying the current request rates with the SLO requirements. Also, our framework auto-scales the number of GPU servers by adapting to the changes in request rates.

Figure 8 presents the overall architecture of our proposed scheduler. The scheduler is composed of a frontend server responsible for making scheduling decisions, and multiple backend servers for executing the decisions. The scheduler on the frontend decides and sends batched requests to the backend servers, and the executor in each backend server dispatches requests to GPUs. The scheduling decision is made by utilizing profiled information of each model (e.g., SLO and inference latency for a pair of batch size and partition size) and incoming request rates. The request monitor tracks the number of newly arrived requests per second for each model. Based on the tracked request rates, the gpulet scheduler decides whether a new organization of partitions is required if the change of request rates is significant enough to update scheduling decisions. If a reorganization is necessary, the new partition ratios are sent to the backend server responsible for

the GPU which needs reorganizing. The partition manager in the backend server prepares the partitions on the GPUs so that they can serve requests with the new partition ratios. The scheduling period is empirically determined based on the GPU partitioning latency to make the overhead of partitioning hidden by the scheduling window.

4.2 Search Space Challenge

The challenge of the three-dimensional scheduling space (batching, temporal, and spatial sharing) for gpulets is that the scheduling decision is affected by several variables dependent on each other. The best size of GPU partition depends on the computational requirement of the model and batch size. Also, the batch size is dependent on the amount of allocated resource and how it is temporally shared with other models to ensure SLO. Therefore, the most cost-effective configuration would sit on the sweet spot in the search space built upon the three dimensions, which creates a huge search space.

To represent the search space, let P be the number of possible GPU partitioning options on a GPU, N be the number of GPUs to schedule, and M be the number of models to serve. Therefore, there are total P^N possible options to partition N GPUs. The M models can be placed on a partition, possibly having all the M models on a single partition. Since we need to check if the consolidation of multiple models violates the SLO, we must evaluate at most M^2 model placements per partition to assess schedulability. As we have NP partitions on the system, the possible mappings of M models to the GPUs is NPM^2 . The complexity of search space is as follows:

$$\text{Total Search Space} = O(P^N NPM^2)$$

As the search space is prohibitively large, it is impractical to exhaustively search and pick a solution. To address the problem, we take a greedy approach, which effectively reduces the search space by allocating partitions to gpulets on GPU incrementally.

4.3 Elastic Partitioning Algorithm

This section discusses our scheduling algorithm called *elastic partitioning* which finds an efficient set of gpulets for given ML inference tasks.

Elastic partitioning: Algorithm 1 describes the overall procedure of scheduling ML models to gpulets. Table 1 lists the variables used in the algorithm. The algorithm receives the following inputs for each model: (1) $L(b, p)$: profiled execution latency of batch size b on partition size p , (2) $intf$: interference function, (3) SLO : per-model SLO in latency, and (4) $gpulet.size$: size of partition allocated for *gpulet*. For every scheduling period, the server checks the request rates of each model. If rescheduling is required, the scheduler performs elastic partitioning with provided inputs (*line 1*).

Name	Description
$L(b,p)$	Latency function of batch size b and partition size p
$intf$	Interference overhead function
SLO_i	SLO (in latency) of model i
$gplet.size$	Actual partition size of gplet

Table 1: Definitions of variables for Algorithm 1.

Algorithm 1: Gplet Scheduling Algorithm

```

ELASTICPARTITIONING( $L(b,p)$ ,  $intf$ ,  $SLO$ ):
1 for each period do // If rescheduling is required
2   Sort every model by  $rate_m \times SLO_m$  in ascending order
3   for each model  $m$  do
4     while ISREMAINRATE() and ISREMAINGPULET()
5       do
6          $rate \leftarrow$  Remaining rate of model  $m$ 
7          $p_{eff} \leftarrow$  MAXEFFICIENTPARTITION()
8          $p_{req} \leftarrow$  MINREQUIREDPARTITION( $rate$ )
9          $p_{ideal} \leftarrow$  MIN( $p_{eff}$ ,  $p_{req}$ )
10         $gplet \leftarrow$  FINDBESTFIT( $p_{ideal}$ ,  $SLO_m$ ,  $intf$ )
11        Apply  $gplet$  to system
12      end
13    end
14  end
15  FINDBESTFIT( $p_{ideal}$ ,  $SLO_m$ ,  $intf$ ):
16  Sort every remaining gplets by size in ascending order
17  for  $gplet$  in GETREMAINGPULETS() do
18    if  $gplet.size \geq p_{ideal}$  then
19      if  $gplet$  is unpartitioned then
20        Split and allocate  $gplet$  to  $p_{ideal}$  size partition
21      end
22       $b \leftarrow \operatorname{argmax}_{k \in \mathbb{N}} (L(k, gplet.size) + intf \leq SLO)$ 
23      if  $b$  exists then
24        TEMPORALSCHEDULING( $gplet$ )
25        return  $gplet$ 
26      end
27    end
28  end

```

Each model is sorted in ascending order by $rate \times SLO$, which corresponds to the amount of work needed for the model (line 2). We allocate starting from the model with the least amount of work to the model which requires the most amount of work as a heuristic optimization for allocating resources. For each model m , the scheduler allocates one or more gplets until the incoming rate can be satisfied or no more gplet is left in the system (line 3-4).

Determining the most effective gplet size: Based on the observation from Section 3.1, the scheduler maximizes the system-wide throughput by allocating the most cost-effective size for gplet.

p_{eff} is the partition size that yields the highest performance per resource, which is the knee point in Figure 4. It is determined during profiling. p_{req} is the partition size satisfying SLO with the batch size that can handle the input rate. When request rates are low, p_{req} can be smaller than p_{eff} .

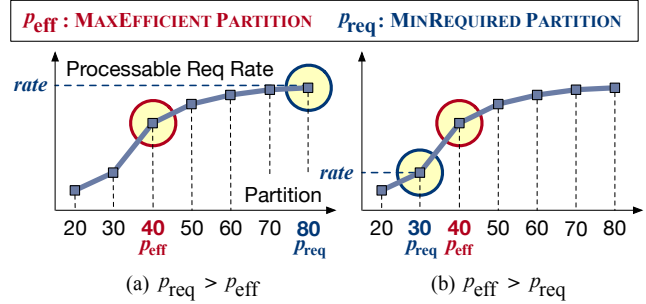


Figure 9: Max efficient partition (p_{eff}) and min required partition (p_{req}).

The scheduler chooses p_{eff} as the best partition size unless p_{req} is smaller than p_{eff} . If p_{req} is smaller than p_{eff} , p_{req} is chosen for the partition size, not to overprovision the GPU resource. Note that for a given model, a matching batch size is fixed for the p_{eff} of the model. Figure 9 presents two cases of p_{eff} and p_{req} . In the algorithm, MAXEFFICIENTPARTITION calculates a sweet spot of profiled gplet size and uses the gplet size at the knee as p_{eff} (line 6). MINREQUIREDPARTITION examines the minimum size of gplet, p_{req} , which is necessary to support the SLO on under the given request rate (line 7). The scheduler picks the minimum of p_{eff} and p_{req} as the ideal partition size p_{ideal} to ensure cost-effective gplet size (line 8).

Incremental allocation with best-fit: After finding p_{ideal} , FINDBESTFIT performs a best-fit search. First, the scheduler sorts the remaining gplets by partition size in ascending order (line 14). The algorithm searches through remaining gplets until a $gplet.size$ is greater or equal to p_{ideal} (line 16). Since the gplets are sorted in ascending order, the sweeping naturally guarantees the best-fit. If the partition of gplet can be split, which means the chosen gplet has a size of 100%, the gplet is split into two gplets, each with a size of p_{ideal} and $100 - p_{ideal}$ (line 17-19). The maximum batch size b is decided and checked whether it can meet the SLO when there is additional interference-induced overhead (line 20). If a valid batch size exists, then the gplet is chosen (line 21).

Temporal scheduling for gplets: After a $gplet$ is chosen, elastic partitioning attempts temporal scheduling between the returned $gplet$ and previously allocated gplets in the system (line 22). Temporal scheduling for gplets follows the same rules which is introduced in Section 2.2: 1) adjust the duty cycle and batch size accordingly, and 2) check whether the SLO can be guaranteed for all models. We introduce an additional rule to consider $gplet.size$ when calculating the batch size and duty cycle. For every pair of gplets, the aforementioned rules are applied to see if temporal sharing is available. If a pair of gplets has a different size, the larger size will be chosen to check if the SLO can be successfully guaranteed or not. If successful, two gplets are merged to a single gplet, thus reducing the total number of required gplets. The scheduler updates the remaining and allocated

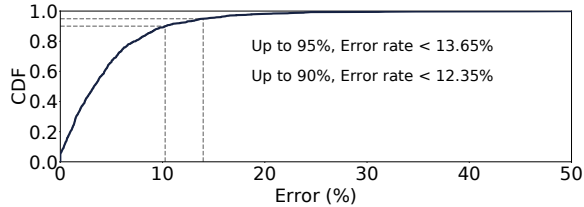


Figure 10: Cumulative distribution of relative error rate. Proposed analytical model can predict up-to 95% of cases with less than 13.98 % error rate.

gpulets with the result of `FINDBESTFIT` (line 10).

Reduced Search Space: Because the cost of iterating through every possible strategy is not required, the search space introduced in Section 4.2 is reduced as follows.

$$\text{Reduced Search Space} = O(NPM^2)$$

Instead of searching every case of P possible partitioning, for every N GPU in the system, each GPU is partitioned incrementally and the cost of checking temporal scheduling for M models still remains. Our search algorithm practically approximates the ideal one and removes P^N from the full search complexity. As a cost of the approximation, it may not always produce a theoretical optimal result. However, our evaluation shows that the algorithm performs closely to the ideal one as presented in Figure 16 (Section 5).

4.4 Modeling Interference

A key challenge in interference handling is to predict latency increases when multiple inferences are executed in different gpulets of the same GPU. As shown in Figure 7, the interference effects are modest for the majority of consolidated executions, yet the overhead could be significant in a few cases.

To confine the interference effect, we provide a simple yet effective interference-prediction model based on two key runtime behaviors of GPU executions. The interference effects of spatial partitioning are commonly caused by the bandwidth consumption in internal data paths including the L2 cache and the external memory bandwidth. To find application behaviors correlated to the interference effects, we profile the GPU with concurrent ML tasks with an NVIDIA tool (Nsight-compute). Among various execution statistics, *L2 utilization* and *DRAM bandwidth utilization* are the most relevant factors correlated to the interference.

Based on the observation, we build a linear regression model with the two parameters (L2 utilization and DRAM bandwidth utilization) as follows:

$$\text{interference_factor} = c_1 \times L2_{m_1} + c_2 \times L2_{m_2} + c_3 \times \text{mem}_{m_1} + c_4 \times \text{mem}_{m_2} + c_5$$

$L2_{m_1}$ and $L2_{m_2}$ are L2 utilization of model m_1 and m_2 , when they are running alone with a given percentage of GPU re-

Algorithm 2: GPU Scaling Algorithm

```

SCALING(GPU_LIMIT):
1 for each period do
2    $N \leftarrow$  The number of used GPUs in previous period
3    $result \leftarrow$  ELASTICPARTITIONING with  $N$  GPUs
4   while  $result$  is fail and  $N < GPU\_LIMIT$  do
5      $N \leftarrow N + 1$ 
6      $result \leftarrow$  ELASTICPARTITIONING with  $N$  GPUs
7   end
8   if  $result$  is fail then
9     Report an unschedulable event
10  end
11 end

```

source. mem_{m_1} and mem_{m_2} are memory bandwidth consumptions of model m_1 and m_2 . Parameters (c_1 , c_2 , c_3 , c_4 , and c_5) are identified by running the linear regression.

We have profiled total 1,250 pairs (total 2,500 data) of inference interference and recorded how much interference each inference task has received. Among 2,500 data, we have randomly selected 1,750 data of execution as training data and 750 data for validation. Figure 10 presents the cumulative distribution of the prediction error with our interference model. The proposed model can predict up to 90% of cases within 10.26% error rate and up to 95% if 13.98 % of error is allowed.

Linear regression is chosen for its relatively high accuracy and low model construction complexity, so it satisfies our purpose in scheduling. Several prior studies have also used such linear models for predicting interference overheads [5, 43, 47].

4.5 Scaling GPUs for Request Rate Changes

During a scheduling period, the monitor tracks the request rates of all ML tasks. If the rates change, it triggers the rescheduling procedure. The rescheduling procedure checks whether the changed rates can be sustained by the current number of GPUs. If not, it tries to increase the number of GPUs to support the SLOs for the new rates. Algorithm 2 presents the rescheduling procedure. It first attempts to use the same number of GPUs of the previous scheduling period (line 2-3). If the *result* fails due to the insufficient number of GPUs, the elastic partitioning is repeated with one additional GPU. However, when the number of required GPUs exceeds the given limit, it reports that an unschedulable event occurs.

4.6 Implementation

SW prototype: The SW prototype of our scheduler was developed in C++ and the approximate lines of code is 20.7K. We have chosen PyTorch for implementing ML inference due to its wide adoption in ML communities, in addition to the readiness to use C++ interfaces.

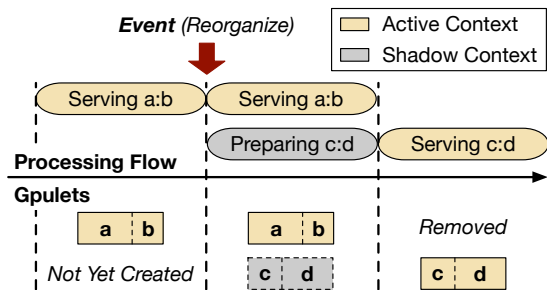


Figure 11: Illustration of dynamic partition reorganization.

Dynamic partition reorganization: NVIDIA provides the MPS control daemon, which allows users to control the proportion of computing resource reserved for processes spawned by the user. The amount of reserved resources is fixed when a process is created. Therefore, To change the proportion of reserved resource, a new process must be created with a new designated amount. This limitation affects our rescheduling procedure with a high cost of adjusting gpulets.

Our scheduling prototype controls gpulet partition size, by spawning a new proxy process to allocate a different amount of partition to gpulet. Preparing a new partition includes spawning a new process, loading kernels used by PyTorch, loading required models, and warming up. As illustrated in Figure 11, to hide the overhead of preparing new partitions when reorganizing is necessary, we overlap the procedure of preparing new partitions with serving the current partitions. The scheduling period for reorganization is 20 seconds which is a conservative estimate of time required for preparing a new partition.

5 Evaluation

5.1 Methodology

Inference serving system specifications: Table 2 provides a detailed description of the evaluated inference system and the used GPU specification. We use two identical multi-GPU inference servers, each of which is equipped with two NVIDIA RTX 2080 Ti GPUs supporting post-Volta MPS capabilities. The table also provides the versions of the operating system, CUDA, NVIDIA drivers, and machine learning framework.

Each GPU server operates as a backend server responsible for executing inference queries on two GPUs. One server additionally generates inference requests while the other server assumes the role of a frontend server to manage backend servers and make scheduling decisions. Both servers are network-connected, imitating inference serving system architecture, with 10 Gbps bandwidth.

Baseline scheduling algorithms: For our baseline, we have ported the Squishy bin-packing (SBP) algorithm (from Nexus [38]) and greedy best-fit introduced in Section 2. We evaluate two versions of our proposed algorithm, gpulet +int

System Overview	
CPU	20-core, Xeon E5-2630 v4
GPU	2 × RTX 2080 Ti
Memory Capacity	192 GB DRAM
Operating System	Ubuntu 18.04
CUDA	10.2
NVIDIA Driver	440.64
ML framework	PyTorch 1.10
GPU Specification	
CUDA cores	4,352
Memory Capacity	11 GB GDDR6
Memory Bandwidth	616 GB/sec

Table 2: The evaluated system specifications.

Model	Input Data (Dimension)	SLO (ms)
GoogLeNet (goo)	ImageNet (3x224x224)	66
LeNet (le)	MNIST (1x28x28)	5
ResNet50 (res)	ImageNet (3x224x224)	108
SSD-MobileNet (ssd)	Camera Data (3x300x300)	202
VGG-16 (vgg)	ImageNet (3x224x224)	142
MnasNet (nas)	ImageNet (3x224x224)	62
Mobilenet_v2 (mob)	ImageNet(3x224x224)	64
DenseNet (den)	ImageNet(3x224x224)	202
Base Bert (be)	Rand. Index Vector(1x14)	22

Table 3: List of ML models used in the evaluation.

considers interference overhead while gpulet does not.

We do not provide a direct comparison to Nexus [38] due to the following reasons: 1) Nexus deploys optimizations that are orthogonal to our work, and 2) several benchmarks that Nexus used in evaluation were not interoperable with our prototype server, as the models are not supported by PyTorch. However, we deploy the same type of video processing models that Nexus used to evaluate the system and show how spatially partitioning GPUs can further enhance performance.

ML models: Figure 12 delineates the detailed dataflow graph of the applications that contain ML models as well as the input/output data. The *game* application analyzes the digits and images from the streamed video games by using seven models in parallel. The *traffic* application is a traffic surveillance analysis with two phases, which are object detection and image recognition. The SLO latency is set as 108 ms and 202 ms for game and traffic, respectively. Each SLO latency is calculated by doubling the longest model inference latency.

Deeper look into particular request scenarios: We choose five model-level scenarios to take a deeper look into the multi-model inference serving. These five scenarios are characterized by the member of models and each respective memory footprint. Table 4 shows the details of each scenario.

Request arrival rate: We sample inter-arrival time for each model from a Poisson random distribution, based on previous literature [48] claiming real-world request arrival rates resemble a Poisson distribution.

Evaluation of request scenarios and applications: For a given scenario or application, we evaluate the scheduling decisions by deploying scheduling results on our prototype

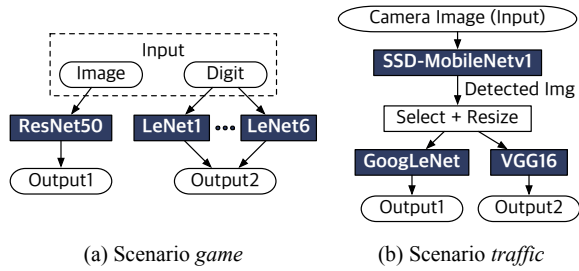


Figure 12: Two multi-model applications: *game* and *traffic*.

Name	Group Composition by Memory Footprint		
	<1GB	1GB - 2GB	>2GB
scen1	mob,be	nas,goo	-
scen2	-	den	vgg
scen3	mob	res	vgg
scen4	ssd	nas,den	-
scen5	le	ssd,nas	vgg

Table 4: Five request scenarios, each of which represents a particular composition of multiple models based on memory footprint. The amount of requests per model in a group is equal across the models in the group.

servers and measuring the throughput under SLOs and SLO violation rates. To consider the unpredictable performance variations, we iterate the experiment three times for each scenario and application, and pick the median result.

5.2 Experimental Results

SLO preserved max throughput: We first evaluate the throughput implications of our schedulers. The *SLO preserved max throughput* is defined to be the maximum achievable throughput while 99% of requests are processed within the SLO latency. We measure the SLO preserved max throughput of the schedulers by gradually increasing the request rate until SLO violation rate exceeds 1% of total requests.

Figure 13 reports the SLO preserved max throughput for the two multi-model applications and five scenarios for four different scheduling algorithms. Our proposed *gpulet +int* scheduler offers higher throughput than both algorithms SBP and greedy best-fit by an average of 61.7% and 81.2%, respectively. Additionally, considering interference yields 7.5% better throughput on average. Although the benefit may seem marginal, we argue that such caution is necessary since a scheduler must be able to guarantee SLO at all times.

The low performance of greedy best-fit is caused by the lack of effective temporal sharing. In Figure 4, models show diminishing returns (over increasing GPU partitions) beyond a knee point. Note that different batch sizes can have different knee points. The greedy best-fit chooses the maximal batch size which satisfies SLOs and sets the partition for the batch size. The spatio-temporal scheduling can select the batch size and partition to better utilize GPU by considering smaller

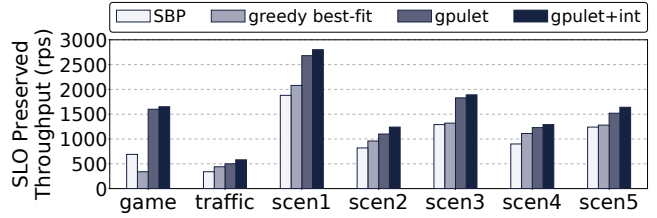


Figure 13: SLO preserved max throughput of the two multi-model applications (*game* and *traffic*) and five scenarios.

batches with temporal sharing across models. The limitation of greedy best-fit is clearly shown in *game*. Its flow has many small parallel tasks (LeNets), which cannot fill even a small GPU partition efficiently. Note that batch sizes cannot be increased arbitrarily as request rates and SLO limits it.

The reported throughput improvement is achievable merely through the MPS features already available in the most server-class GPUs and scheduling optimization in software, using the same GPU machine. Thus, by utilizing otherwise wasted GPU resources, the proposed scheduling scheme would be able to virtually offer cost savings for the ML inference service providers. For instance, *gpulet +int* achieves 1,650 req/s throughput for *game* while SBP does 690 req/s, utilizing the identical physical system, which can be translated into 58.2% effective cost saving ($= \{1 - \frac{690}{1650}\} \times 100$).

The effect of interference model: This analysis shows how the interference model can avoid SLO violations by correctly incorporating the interference effect into the scheduling decision. In this result, we measured SLO violation rates by gradually increasing request rates until both *gpulet* and *gpulet +int* consider the current rate not schedulable. Figure 14 presents the SLO violation rates when the system is receiving the maximum request rate before both of them reach the not schedulable decision. In the figure, if the violation rate is higher than 1%, the case is highlighted with a red round. The scheduler *gpulet*, which does not consider interference, shows violation rates higher than 1% even for the rates that it considered to be schedulable for *scen2*, *scen3*, and *scen5*. However, *gpulet +int* successfully filters out such rate by either classifying as *not schedulable (N)* as shown in *scen2* and *scen3* or successfully scheduling tasks without violating the SLO such as *scen5*.

Evaluation of scalability: To evaluate whether our prototype scheduler can successfully scale *gpulets* to accommodate fluctuating rates, we measure the performance of our scheduler while submitting inference requests with varying rates for all models in *scen3*. We have chosen *scen3* because of its evenly distributed model size to reproduce a realistic workload.

To evaluate scalability beyond our testbed, we launched multiple servers by running each server with docker container. By running one container per GPU with four more identical GPUs, we conducted our experiment on total eight servers. Additionally, the request generator and frontend server were specially tailored to send dimensional data, instead of actual

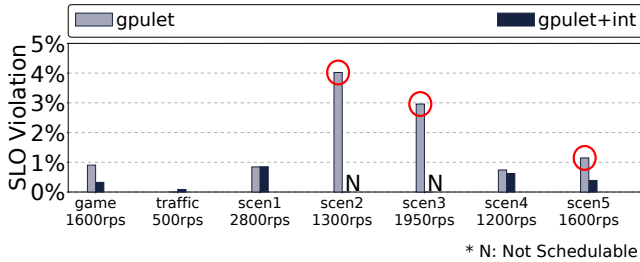


Figure 14: SLO violation rates of two multi-model applications and five scenarios. Request rates are increased until both gpulet and gpulet+int concluded the rate to be *Not Schedulable*.

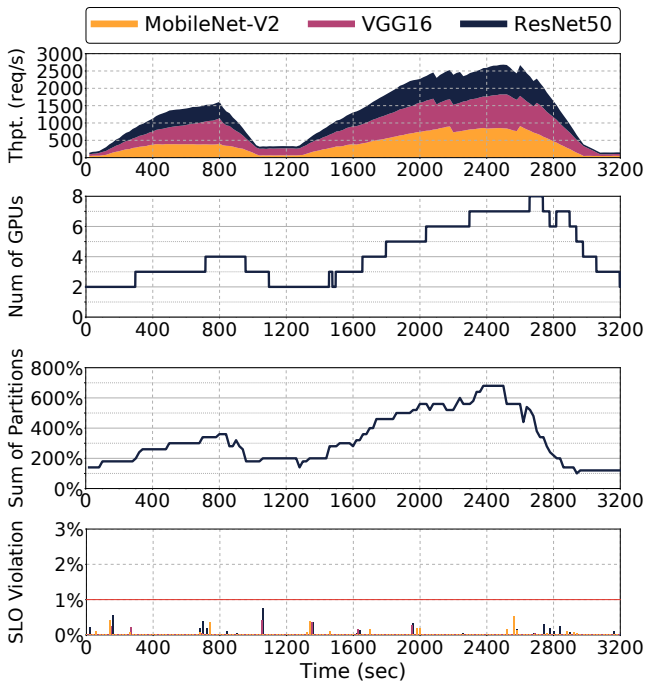


Figure 15: (a) Throughput(req/s), (b) number of GPUs, (c) sum of partition size (%) of gpulet, (d) SLO violation (%) of each model for 3,200 seconds.

data for inference request, for overcoming network bottleneck between physical servers. The backend servers behave identically other than generating random data with dimensions provided from frontend server. Figure 15 reports how our scheduling framework performed for a 3,200 second window. The top graph shows a stacked graph of the accumulated throughput of each model. The second and third graph reports how many GPUs were scheduled and the sum of partition sizes of gpulets, respectively. The last graph depicts the percentage of SLO violation (including dropped requests) for 20 second period. Between 0 and 1,200 seconds, the rate gradually increases and decreases to its initial rate. As the rate rises, our proposed scheduler successfully allocates more GPUs to preserve SLO. When the rate decreases, the sum of utilized partitions also decreases by reorganizing partitions. The following wave, starting from 1,400 seconds, rises to a

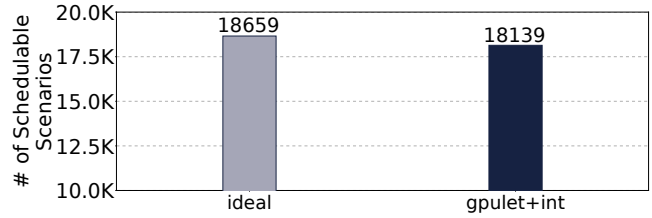


Figure 16: Comparison of the numbers of schedulable scenarios between the ideal scheduler and gpulet +int scheduler.

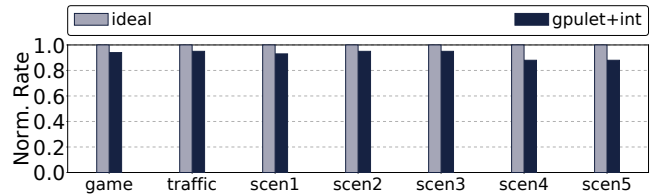


Figure 17: Comparison of the normalized maximum schedulable rates with real-world multi-model benchmarks and five scenarios.

higher peak than the previous wave of requests. Nonetheless, our scheduler successfully adjusts gpulets and preserve SLO, guaranteeing SLO violation rate lower than 1%.

Comparison to the ideal scheduler: We evaluate the scheduling capability of elastic partitioning by comparing the scheduling results produced from an ideal scheduler. To produce various model-level inference request scenarios, we use the same methodology described in Section 3.2, which populates a set of 19,682 possible scenarios. The ideal scheduler makes scheduling decisions by exhaustively trying all 4^4 partition combinations, where 4 GPUs can be partitioned into either (2:8), (4:6), (5:5), or (10:0). The search continues until all cases are searched or a case produces a viable scheduling result for a given request scenario. For a fair comparison, the ideal scheduler uses the same set of partitions as gpulet+int. Figure 16 compares the number of scenarios classified as schedulable by each scheduler. gpulet+int can schedule 520 fewer cases compared to ideal, which is 2.6% of the total 19,682 cases.

Figure 17 reports the maximum schedulable rate of each multi-model scenario. All rates are normalized to the maximum rate which ideal can provide. gpulet+int can achieve an average 92.6% of the maximum rate which the ideal scheduler can provide.

6 Related Work

6.1 Prior ML Systems Studies

Machine learning service platforms: A wide variety of computer systems and researches have been proposed to improve the quality of machine learning services [1, 2, 9, 13, 15,

Features	Batch Tuning	Multi Model	GPU Scaling	Temporal Schedule	Spatial Schedule	Interference Prediction
Clipper [15]	✓	✓	✓	✓	✗	✗
MArk [45]	✓	✗	✓	✗	✗	✗
INFaaS [36]	✓	✓	✓	✓	✗	✗
Nexus [38]	✓	✓	✓	✓	✗	✗
GSLICE [17]	✓	✓	✗	✗	✓	✗
Gpulet	✓	✓	✓	✓	✓	✓

Table 5: Comparison with prior work.

17, 21, 22, 24, 26, 32, 35, 36, 37, 38, 39, 40, 41, 45]. INFaaS is a platform for serving inference that guarantees SLO and minimizes the cost by choosing an adequate variation of a model [36]. INFaaS adopts a reactive approach when dealing with interference caused by co-locating model variants on the same hardware resource. Clockwork focuses on providing an accurately predictable system by leveraging the fact that the latency of inference is relatively consistent [21]. Clockwork preferred predictability over utilization gains from co-locating models and thus does not consider spatial sharing.

Although this paper did not cover training, past research inspired this study with schedulers for optimizing GPU resource [16, 24, 33, 42, 46]. Another related research direction focus on how to ease the burden of deployment and optimization for machine learning across various platforms [6, 27, 28, 31, 32]. Prior studies related to cluster scheduling have also influenced this paper [20, 30, 43].

Interference estimation: Precise estimation of interference has been a key issue for high-performance computing. Bubble-up [29] and bubble-flux [44] models an application’s sensitivity to cache and fits a sensitivity curve to predict performance. Han *et al.* extend using sensitivity cure to distributed computing where interference can propagate among processes [23]. Prophet models concurrent task execution behavior for non-preemptive accelerators [4].

Multi-tenancy in Accelerator: GPU vendors have included HW/SW support for providing multi-tenancy to users such as NVIDIA Multi Process Service (MPS) [12], Multi Instance GPU (MIG) [14], and AMD MxGPU [10]. Academic researches also proposed multi-tenancy support in accelerators. Pratheek *et al.* devised page-walking stealing for multi-tenancy support in GPU [34]. Choi *et al.* proposes fine-grain batching scheme [8]. PREMA proposed time-multiplexing solution with preemption [7]. Planaria supports multi-tenancy by partitioning processing elements [19].

6.2 Comparison to Prior Work

Table 5 provides a summarized comparison of our work to related ML inference frameworks. All the prior studies are capable of dynamically tuning batch size by either leveraging profiled latencies or incoming request rates during runtime. As more ML workloads are consolidated in cloud-based GPU servers, scheduling of multiple heterogeneous ML models in a system and scaling GPU servers under fluctuating request rates become more important. However, some prior studies

do not consider such multi-model supports or GPU scaling.

Regarding scheduling dimensions such as temporal and spatial sharing, a majority of the prior work employ temporal sharing by leveraging profiled information of latency. Only GSLICE considered spatial sharing but it does not consider multi-GPU scheduling and temporal sharing. On the other hand, our study addresses all challenges, scheduling dimensions, and predicting potential interference among partitions in the same GPU.

7 Conclusion

This study investigated an SLO-aware ML inference server design. It identified that common ML model executions cannot fully utilize GPU compute resources when their batch sizes are limited to meet the response time-bound set by their SLOs. By leveraging spatial partitioning features, our framework significantly improved throughput of multi-GPU configurations while supporting SLOs. Based on the new spatio-temporal scheduling technique, this study showed that a new abstraction of GPU resources (*gpulet*) can improve ML inference serving under SLOs. The source code is available at <https://github.com/casys-kaist/glet>.

8 Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF-2022R1A2B5B01002133) and Institute of Information & communications Technology Planning & Evaluation (IITP) grants funded by the Ministry of Science and ICT, Korea (IITP2017-0-00466, IITP2021-0-01817).

References

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
- [2] Ahsan Ali, Riccardo Pincioli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020.
- [3] E. Baek, D. Kwon, and J. Kim. A multi-neural network acceleration architecture. In *2020 ACM/IEEE 47th*

Annual International Symposium on Computer Architecture (ISCA), 2020.

- [4] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
- [5] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.
- [6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
- [7] Y. Choi and M. Rhu. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.
- [8] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. Lazy batching: An sla-aware batching system for cloud machine learning inference. *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.
- [9] Amazon Corporation. *Amazon SageMaker Developer Guide*, 2020. <https://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-dg.pdf>.
- [10] AMD Corporation. *AMD MULTIUSER GPU:HARDWARE-ENABLED GPU VIRTUALIZATION FOR A TRUE WORKSTATION EXPERIENCE*, 2016. <https://www.amd.com/system/files/documents/amd-mxgpu-white-paper.pdf>.
- [11] NVIDIA Corporation. *Deep Learning Inference Platform*, 2013. <https://www.nvidia.com/en-us/deep-learning-ai/solutions/inference-platform/>.
- [12] NVIDIA Corporation. *Multi-Process Service*, 2019. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.
- [13] NVIDIA Corporation. *TensorRT Developer's Guide*, 2020. <https://docs.nvidia.com/deeplearning/sdk/pdf/TensorRT-Developer-Guide.pdf>.
- [14] NVIDIA Corporation. *NVIDIA Multi-Instance GPU User Guide*, 2021. https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_MIG_User_Guide.pdf.
- [15] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.
- [16] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-Specialized Parameter Server. In *Proceedings of the 11th European Conference on Computer Systems (Eurosys)*, 2016.
- [17] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. Gslice: Controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*, 2020.
- [18] Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.
- [19] S. Ghodrati, Byung Hoon Ahn, J. K. Kim, Sean Kinzer, Brahmendra Reddy Yatham, N. Alla, H. Sharma, Mohammad Alian, E. Ebrahimi, Nam Sung Kim, C. Young, and H. Esmaeilzadeh. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks. *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.
- [20] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM conference on Special Interest Group on Data Communication (SIGCOMM)*, 2014.
- [21] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
- [22] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. Cocktail: A multidimensional optimization for model serving in cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.
- [23] Jaеung Han, Seungheun Jeon, Young ri Choi, and Jaehyuk Huh. Interference Management for Distributed

- Parallel Applications in Consolidated Clusters. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.
- [24] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. DjiNN and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.
- [25] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.
- [26] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic Space-Time Scheduling for GPU Inference. In *Proceedings of the Conference and Workshop on Neural Information Processing Systems (NeurIPS)*, 2018.
- [27] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel GPU task scheduling for deep learning. In *Proceedings of the Conference and Workshop on Neural Information Processing Systems (NeurIPS)*, 2020.
- [28] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
- [29] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.
- [30] Shanka Subhra Mondal, Nikhil Sheoran, and Subrata Mitra. Scheduling of time-varying workloads using reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2021.
- [31] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
- [32] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-Serving: Flexible, High-Performance ML Serving. In *Proceedings of the Conference and Workshop on Neural Information Processing Systems (NeurIPS)*, 2017.
- [33] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the 13th European Conference on Computer Systems (Eurosys)*, 2018.
- [34] B Pratheek, Neha Jawalkar, and Arkaprava Basu. Improving gpu multi-tenancy with page walk stealing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.
- [35] Kiran Ranganath, Joshua D Suetterlein, Joseph B Manzano, Shuaiwen Leon Song, and Daniel Wong. Mapa: Multi-accelerator pattern allocation policy for multi-tenant gpu servers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.
- [36] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. Infaas: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, July 2021.
- [37] Wonik Seo, Sanghoon Cha, Yeonjae Kim, Jaehyuk Huh, and Jongse Park. Slo-aware inference scheduler for heterogeneous processors in edge platforms. *ACM Trans. Archit. Code Optim.*, 2021.
- [38] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.
- [39] Chengcheng Wan, Muhammad Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, and Shan Lu. Alert: Accurate learning for energy and timeliness. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.
- [40] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. Morphling: Fast, near-optimal auto-configuration for cloud-native model serving. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2021.
- [41] Wei Wang, Sheng Wang, Jinyang Gao, Meihui Zhang, Gang Chen, Teck Ng, and Beng Ooi. Rafiki: Machine Learning as an Analytics Service System. In *Proceedings of the 44th International Conference on*

Very Large Data Bases (VLDB), 2018.

- [42] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
- [43] Ran Xu, Subrata Mitra, Jason Rahman, Peter Bai, Bowen Zhou, Greg Bronevetsky, and Saurabh Bagchi. Pythia: Improving datacenter utilization via precise contention prediction for multiple co-located workloads. In *Proceedings of the 19th International Middleware Conference*, 2018.
- [44] Hailong Yang, Alex D. Breslow, Jason Mars, and Lingjia Tang. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.
- [45] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2019.
- [46] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2017.
- [47] Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.
- [48] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

A Artifact Appendix

A.1 Abstract

To maximize the resource efficiency of inference servers, we proposed a key mechanism to exploit hardware support for spatial partitioning of GPU resources. With the partitioning mechanism, a new abstraction layer of GPU resources is created with configurable GPU resources. The scheduler assigns requests to virtual GPUs, called gpulets, with the most effective amount of resources. The prototype framework auto-scales the required number of GPUs for a given workloads, minimizing the cost for cloud-based inference servers. The prototype framework also deploys a remedy for potential interference effects when two ML tasks are running concurrently in a GPU.

A.2 Hosting

The artifact is hosted on the following platforms:

- **Zenodo:** We have published the artifact on Zenodo: <https://doi.org/10.5281/zenodo.6544909>
- **GitHub:** Although the artifact provided in Zenodo contains all necessary and functional code, it is still in its early stage of development and needs improvement in terms of UI and code readability. Further improvement of code will be provided in the following GitHub repository: <https://github.com/casys-kaist/glet>.

A.3 Scope

The artifact is capable of:

- Serving machine learning inference on multiple multi-GPU servers.
- Adding new models defined and saved by TorchScript (provided as .pt files).
- Scheduling multiple models and guarantee SLO.
- Providing stand-alone ML inference executor for profiling performance and GPU resource usage.

The artifact does *not* provide the following:

- Utilize CPU for ML inference.
- Schedule heterogeneous GPUs.
- Guarantee availability (e.g. heartbeat, failure recovery).
- Add new models or deleting old models to serve while the frontend server is running.

A.4 Contents

A.4.1 SW Components

- **Frontend Server:** Receives requests from M clients and schedule requests to N backends.

- **Backend Server:** Receives batched requests from frontend server and conveys request to 0 servers running on the same machine.
- **Proxy:** Receives inputs from backend server and executes ML inference on a gpulet
- **Standalone Inference:** Executes inference on a GPU. Useful for debugging and profiling GPU resource utilization.
- **Standalone Scheduler:** Provides scheduling decision for given set of models and input rate of each model as stand-alone SW. Useful for inspecting scheduling decisions.

Please refer to *binaries.md* for further information of how to run and setup each components.

A.4.2 Models

The artifact includes 0 CNN models of VGG16 and ResNet50. Both models are stored as .pt file. All models are also available on Torchvision.

A.4.3 Dataset

A subset of ImageNet data and camera surveillance footage are each compressed as imagenet_data.tar and camera_data.tar respectively.

A.4.4 Docker Images

The following prerequisites must be installed in order to use the Docker images for this artifact:

- **Docker Ver.** ≥ 20
- **Nvidia-docker** (for utilizing GPUs)

Two Docker images are made public for experimenting with the provided artifact. One is the server Docker image available on sbchoi:glet-server and the other is the base Docker image used for building the backend Docker image available on sbchoi:glet-base.

We highly recommend using Docker images for experimenting since it contains all required code and scripts. For further instructions, please refer to the README file on <https://github.com/casys-kaist/glet>.

A.5 Requirements

A.5.1 Hardware

The artifact was evaluated on multi-GPU servers. Each GPU server had the following hardware specifications:

- **GPU:** NVIDIA RTX 2080ti (11GB global memory)
- **CPU:** Intel Xeon E5-2630 v4
- **Network:** Servers connected with 10 GHz Ethernet

A.5.2 OS and Kernel

The artifact was evaluated on Ubuntu 18.04 with a Linux kernel version 4.15.

A.5.3 Software

The artifact was built with the following drivers and libraries:

- LibTorch(PyTorch library for C++) = 1.10
- CUDA \geq 10.2
- cuDNN \geq 7.6
- Boost library \geq 1.6
- OpenCV \geq 4.0
- CMake \geq 3.19

A.6 Experiment Setup

Experiments can be run by using the scripts provided in the artifact. We have also provided example files required for configuring experiments. Below are a few steps to configure multiple GPU servers using Docker images we have provided:

1. Run MPS daemon
2. Create and run an overlay network for Dockers
3. Setup and execute backend servers
4. Setup and execute frontend server, connecting all backend servers for serving inference.
5. Run clients
6. Analyze the content of `glet/scripts/log.txt` for how each request has been handled.

Please refer to the *README* file and *binaries.md* in <https://github.com/casys-kaist/glet> for detailed instructions of how to configure

PilotFish: Harvesting Free Cycles of Cloud Gaming with Deep Learning Training

Wei Zhang*
Shanghai Jiao Tong University

Binghao Chen
Shanghai Jiao Tong University

Zhenhua Han
Microsoft Research Asia

Quan Chen
Shanghai Jiao Tong University

Peng Cheng
Microsoft Research Asia

Fan Yang
Microsoft Research Asia

Ran Shu
Microsoft Research Asia

Yuqing Yang
Microsoft Research Asia

Minyi Guo
Shanghai Jiao Tong University

Abstract

Cloud gaming services have become important workloads in cloud datacenter. However, our investigation shows that a cloud gaming service cannot saturate the modern cloud GPUs. One way to improve the GPU utilization is to co-locate multiple workloads within one GPU, which is challenging for cloud gaming due to its highly fluctuated and unpredictable GPU usage pattern. In this paper, we present PilotFish, a high-performance system that harvests the free GPU cycles of cloud gaming with deep learning (DL) training, while incurring almost zero interference to cloud gaming. We co-locate DL training jobs with cloud gaming, because they have stable and predictable workloads and have no strict latency requirement. In more detail, PilotFish captures the idle periods of the game's GPU usage with its low-overhead instrumentation to graphic libraries in sub-millisecond granularity. To avoid the potential interference to cloud gaming, PilotFish schedules training computation kernels only when they can finish before the idle GPU periods, and preempts straggler kernels running longer than expected. Our evaluation on popular cloud games and DL models shows PilotFish can harvest up to 85.1% of the idle GPU time from cloud gaming with no interference.

1 Introduction

Cloud gaming is gaining popularity in recent years. As shown in Figure 1, players of cloud gaming only use a thin client that interacts with games running on cloud servers and receives the stream of rendered frames via Internet [38]. Cloud gaming greatly reduces the hardware requirement of high-quality video games. Mobile clients with no or weak GPU can still enjoy the good visual effect of powerful GPUs. Cloud gaming has become an important workload in major cloud service providers, e.g., Microsoft's Xbox Remote Play [11], Google's Stadia [7], Nvidia's Geforce Now [15], Sony's PlayStation Now (running on Azure) [18], Amazon's AppStream [1].

*This work is done while Wei Zhang is an intern in Microsoft Research.

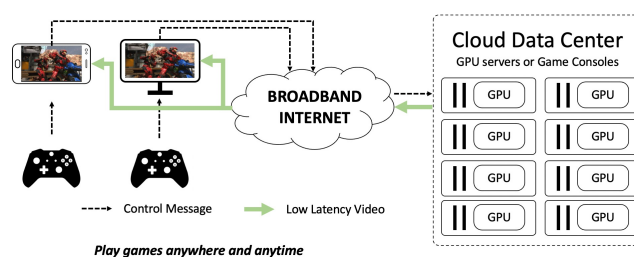


Figure 1: In cloud gaming games, players send control messages (keyboard and mouse) to cloud servers. Game scenes are rendered as frames in cloud servers and streamed to edge devices via internet.

Due to the limitations of the network, encoding and decoding capability, and resolution of mobile devices, major cloud gaming service only provides limited streaming quality that is far lower than the increasing capability of modern GPUs. For example, Microsoft's Xbox Remote Play and PlayStation Now only support up to 1080p at 60FPS. However, the latest GPUs for gaming (e.g., Nvidia's 3090Ti) can support 4K (2160p) resolution at up to 144FPS. Running cloud gaming of limited streaming quality on powerful GPUs would inevitably waste the GPU cycles. Our evaluation of popular games shows most of them have a utilization lower than 50% with cloud gaming GPUs. It is important to improve the utilization to reduce the operation cost of cloud gaming services.

To improve GPU utilization for cloud gaming, a natural solution is to co-locate multiple workloads in one GPU (e.g., multiple games [29, 36] or other GPU workloads [23–25, 50, 51]). Such approaches face great challenges, due to the high randomness of the gaming workload. A game's utilization of different resources (including GPU, CPU, PCI-e and disk I/O) varies greatly across video frames. Such variation is difficult to predict due to the random interaction between players and changing game scenes. Moreover, different games could exhibit very diverse resource usage patterns, further increasing the degree of unpredictability. Co-locating multiple games in a GPU would inevitably lead to interfer-

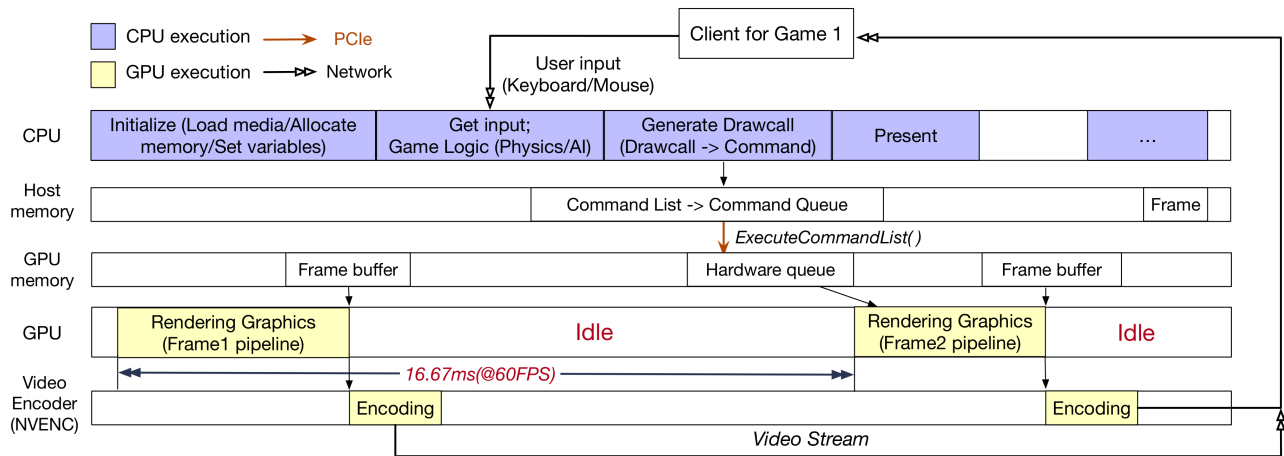


Figure 2: The procedures of cloud gaming. On receiving user input, the game logic decides the content of the scene to be rendered, which is comprised of a list of draw calls using graphic libraries. The draw calls are pushed into a command queue for the frame and submitted to the hardware for rendering the frame. The rendered frames are encoded by dedicated chips (e.g., NVENC [13] of Nvidia’s GPUs) and sent to the cloud gaming client.

ence when long rendering times from different games collide. To safely harvest the GPU free cycles from cloud gaming, it is necessary to choose a more predictable and stable workload for co-location, where we find Deep Learning (DL) training, a pervasive workload in cloud data centers, is a good fit.

In this paper, we present PilotFish, a high-performance system that harvests the free GPU cycles of cloud gaming with deep learning training, without impacting gaming experiences. Instead of predicting the varying gaming workload, PilotFish exploits idle GPU periods in a reactive manner. PilotFish exposes a real-time resource monitoring interface by instrumenting graphic libraries (e.g., DirectX or Vulkan) for quickly reporting (within 10 μ s) the start and completion of the rendering of a game frame. This way, PilotFish can precisely capture idle GPU periods of games. This design allows PilotFish to support all games running on common graphic libraries without modifying or re-compiling game code.

PilotFish further leverages the *predictability* of deep learning training in the scheduling. It is well-known that deep learning training consists of iterative training steps. The compute kernels in each training step have a highly predictable execution time and can be obtained through offline profiling [40,46]. With the known duration of a specific DL training kernel (usually on the order of sub-millisecond), PilotFish is able to safely schedule a deep learning training job to leverage the idle time of gaming workload, without violating the QoS of cloud gaming. The interference on other types of hardware resources is also avoided via state-of-the-art techniques, e.g., Baymax [25] for PCI-e. Furthermore, to prevent from training anomaly, where a DL training kernel does not complete in the estimated time, PilotFish can proactively terminate the training (<1ms) with limited loss of training progress.

We have implemented a prototype of PilotFish to sup-

port games on DirectX 12 [3] and DL training using Nvidia CUDA 11 [14]. We evaluate PilotFish using popular games for cloud gaming and widely-used DL models for training. Evaluation result proves PilotFish can strictly guarantee the QoS of cloud gaming when co-located with DL training. PilotFish can harvest up to 85.1% of the idle GPU time without interference, compared to straw-man baselines that degrade the 99%-ile FPS by over 30% to achieve the same harvest ratio.

The key contributions of the paper are as follows:

- We identify the low GPU utilization problem of cloud gaming and the challenges of co-location due to the randomness of games.
- We characterize the cloud gaming workload and point out that DL training is a right workload to be co-located with cloud gaming to improve GPU utilization.
- We propose mechanisms for quickly capturing idle GPU periods of gaming and fine-grained scheduling of co-located DL training workload, which guarantee no interference.

2 Motivation

In this section, we study the common cloud gaming pipeline shown in Figure 2. We investigate why there is low utilization issue in cloud gaming services and the challenges of harvesting free GPU cycles from games. Then we motivate why DL training is a good fit for co-location.

Table 1: The GPU and CPU utilization of cloud gaming.

Game	Average GPU Util.	Peak GPU Util.	VRAM (GB)	CPU Util.	FPS
Dota 2	38.2%	45%	1.61	21.9%	59.9
League of Legends	26.9%	41%	1.16	22.0%	59.8
PUBG	40.6%	95%	4.05	28.9%	60.1
CS:GO	45.0%	57%	2.6	69.7%	201
Civilization 5	32.3%	42%	1.11	15%	59.8
The Division 2	89.5%	98%	3.12	46.11%	58.66
Assassin's Creed Odyssey	69.2%	78%	2.39	66.3%	59.68
Ashes of the Singularity	89.8%	98%	3.42	79.23%	57.31

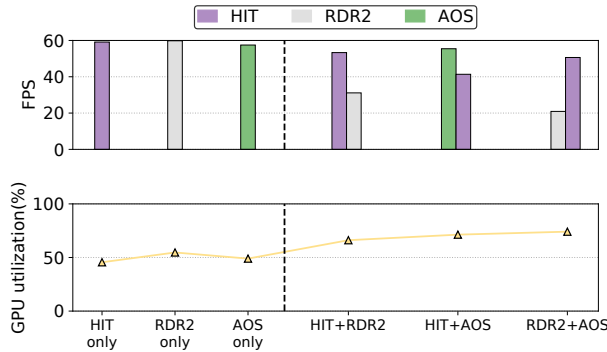


Figure 3: The average FPS and GPU utilization of independent execution of popular games and their co-located execution on Nvidia RTX 2060. (HIT: HITMAN3, RDR2: Red Dead Redemption 2 and AOS: Ashes of the Singularity)

2.1 GPU Under-utilization of Cloud Gaming

Existing cloud gaming platforms allocate each player to a dedicated server for running the requested game to ensure players' satisfactory experiences. For cloud gaming service providers, major concerns are focused on network latency and operational cost. The network latency is considerably reduced today and becomes viable for cloud gaming. However, the low resource utilization still leads to significant operation cost. We use the Nvidia RTX 2060 GPU, of which the computing ability is 6.4 teraflops, as the experimental platform, which has comparable performance to the Xbox One X's GPU (6.01 teraflops) used by Microsoft's cloud gaming service. We investigate the performance of eight of the most popular games. Table 1 summarizes the resource utilization of these games on NVIDIA RTX 2060 with cloud gaming rendering quality, mostly 1080p and 60 Frames Per Second (FPS). Five of the eight games have a GPU utilization of lower than 50%, showing the potential opportunities for improvement.

Modern GPUs are becoming more and more powerful. However, the QoS of cloud gaming is much lower than the capability of modern GPUs. According to Steam's survey [19], over 83.67% of PC gamers use resolution $\leq 1920 \times 1080$. Most smartphones only have a screen $\leq 1080p$ resolution. Also, the higher resolution requires better network quality and hardware capability (for decoding). Currently, Xbox remote play

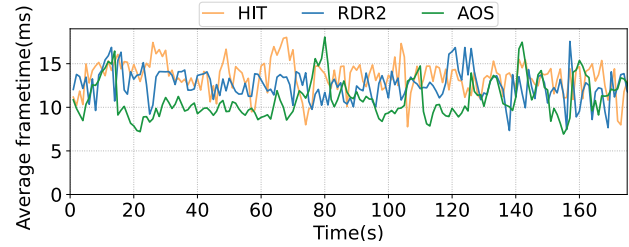


Figure 4: The fluctuation of frame time over time of three popular games.

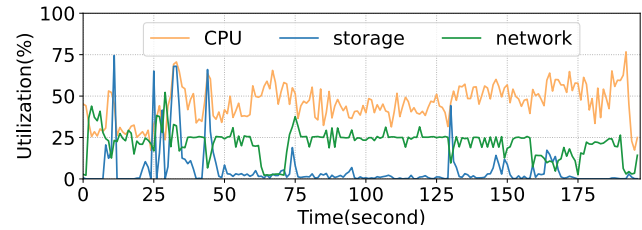


Figure 5: The fluctuation of CPU, storage and network utilization over time of Hitman 3.

only supports streaming quality of at most 1080p and 60 FPS.

We anticipate the low GPU utilization issue will become more severe on the latest generation of GPUs used by gaming clouds. For example, Google Stadia uses an AMD GPU with 10.7 teraflops [7], Microsoft's Xbox Series X chip has 12 teraflops [11], Nvidia's RTX 3090 has 35.58 teraflops.

2.2 Challenges

A natural idea to improve the GPU utilization of cloud gaming is to co-locate multiple games into the same GPU. However, we observe co-locating multiple games could severely interfere with each other, even when the GPU is still underutilized.

Figure 3 demonstrates the FPS of three popular cloud games and their GPU utilization in two situations: independently execution and co-located execution. When the games run alone, they can achieve around 50% of GPU utilization on 60 FPS. However, when two games are co-located, the FPS drops greatly (e.g., RDR2's FPS drops to 20 from 60) but the GPU utilization is only improved by up to 24%.

The main cause of the degraded co-location performance comes from the randomness of games. As shown in Figure 4, a game's frametime (the time to render a frame) could vary significantly over time due to the different complexity of the scene. Different games would further exhibit a very diverse pattern of GPU consumption. Moreover, in addition to GPUs, the resource usage of other resource types (CPU, storage, etc.) also fluctuates over time as illustrated in Figure 5. When contention appears on these resources, the submission of draw calls would be blocked, which also leads to lower GPU utilization. This explains why the co-located games only have

limited improvement on GPU utilization in Figure 3.

The highly random gaming behaviours make it impossible to co-locate the other random and interactive workloads like game without impacting the gaming experience. Previous works [36, 43] using static profiling to co-locate multiple games in a best-effort manner could still suffer from the interference due to random rendering content. We seek to find a more stable and predictable workload as the candidate for co-location, where we find DL training is a good fit.

2.3 Co-location with DL Training

DL training is a pervasive workload in cloud data centers. The major cloud gaming service providers (e.g., Microsoft, Google, Amazon) also have a huge demand for training DL models with GPUs [33]. The key reason we consider DL training for co-location with cloud gaming is its predictability and fine-granularity. Figure 6 shows the execution time of statistical top 20 frequent kernels from six popular DL training models. The figure shows that the duration of all training kernels is relatively stable, it usually varies within a few percent. Thus, by leveraging the predictability and iterative pattern of DL kernels, the system can know their duration *beforehand*. Also, the execution time of DL kernels is typically less than 1ms, thus it is very suitable to be scheduled to exploit the GPU idle time.

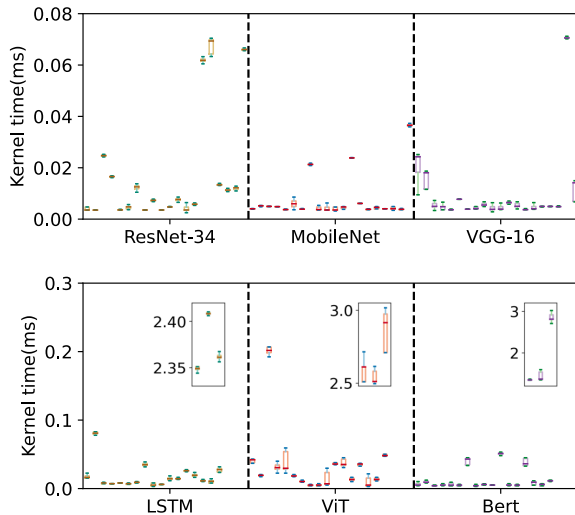


Figure 6: The execution time of top 20 frequent kernels from six popular models. (each bar is a kernel).

Despite the opportunity, the direct co-location of gaming and DL training without leveraging the characteristics of DL training could still incur a severe drop in FPS due to complex interference behaviors. For example, if the DL training kernels are submitted when a frame is still under rendering, both workloads would contend for GPU time and postpone the completion of game rendering.

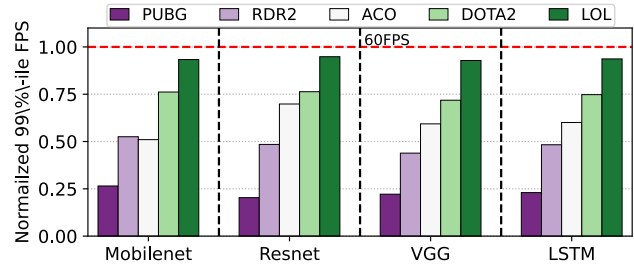


Figure 7: FPS of games when co-located with DL training.

Figure 7 shows the normalized 99%-ile FPS when five popular games are co-located with DL training tasks on a Nvidia 2060 GPU. The DL training tasks includes Resnet50 [30], VGG [44] and Mobilenet [31]. In the figure, the x-axis indicates the combination of games and DL training tasks, and the y-axis shows the 99%-ile FPS normalized to its FPS target. All games are affected by naively co-locating with DL training models. During co-location, we observe that the FPS of game is affected by the *duration of frame rendering*, the *DL training kernel scheduling* and the *contention on shared resources*. Therefore, it requires very careful management of the co-located DL training jobs to avoid interfering the cloud gaming, which is the main goal of PilotFish.

3 PilotFish Overview

We consider the scenario that DL training has a lower priority than the interactive cloud gaming service. Therefore, it is required that DL training should not generate interference to cloud gaming. PilotFish co-designs the cloud gaming services and deep learning training frameworks so that they can collaboratively work together. Figure 8 demonstrates the overall design of PilotFish.

Instead of predicting the random gaming behaviours, *PilotFish monitors the frame-level execution and resource-usage information in real time with very low overheads*. Existing frame monitoring tools [8, 9, 17] for gaming are usually based on event-tracing technology [4], which is for general-purpose application by design and infeasible for PilotFish’s requirement due to its high latency. To capture the idle GPU periods, PilotFish instruments the graphic libraries to quickly and precisely detect when the rendering of a frame finishes and when the next frame will be submitted (according to the FPS requirement).

Within the idle period of a game, *PilotFish schedules the computation kernels safely without interfering with the games*. The computation kernels for DL training should only be executed between the end of the previous frame and the start of the next frame. This relies on the kernel duration predictor to provide the execution time of the computation kernels, by leveraging their predictability as we discussed in Figure 6. A computation kernel can be submitted only when it can finish

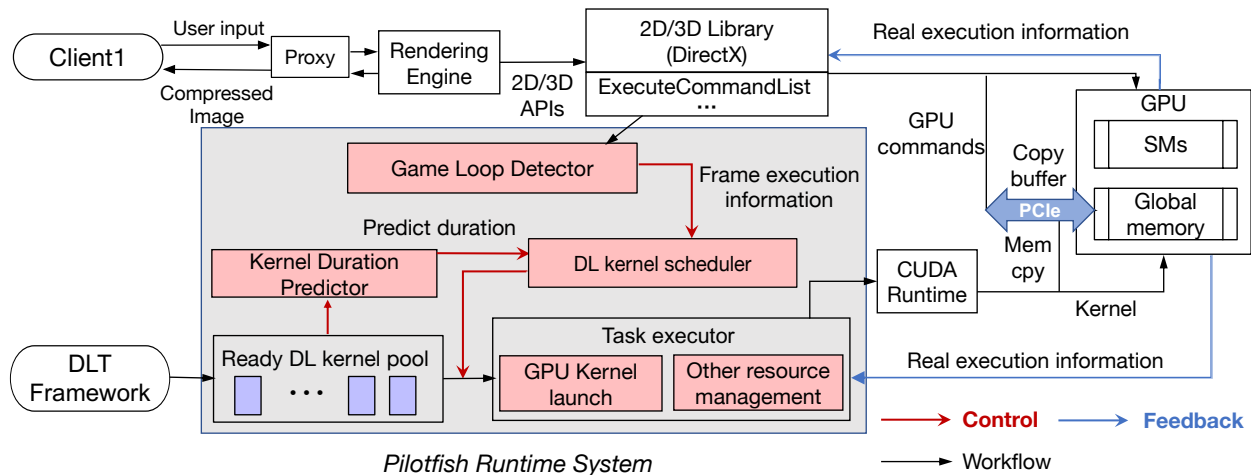


Figure 8: The Overall design of PilotFish. The Game Loop Detector quickly obtains the idle GPU periods via instrumenting the graphic libraries. The DL kernel scheduler dynamically and safely schedules the kernels with the predicted kernel duration. The task executor guarantees the DL kernel execution will not interfere with cloud gaming on GPU and other types of resources.

before the rendering of the next frame starts so that it will not contend with the game rendering on GPU. Since PilotFish only schedules DL kernels without changing its computation, it has no impact on the computation result of DL training.

During the execution of DL training’s computation kernels, PilotFish keeps monitoring their progress in its task executor. Once potential interference could appear due to straggler kernels, *PilotFish should immediately preempt the job to guarantee cloud gaming is not affected*. To minimize the loss of training progress due to preemption, we introduce a low-overhead checkpointing mechanism to only kill the computation kernels without losing the trained weight in memory.

We explain in § 4 how PilotFish instruments the graphic libraries to obtain the idle GPU periods. In § 5, we elaborate on how the computation kernels of DL training are scheduled. Then, we demonstrate the task executor in § 6 that manages the task execution on GPUs and other types of resources to provide the strict guarantee of no interference to games.

4 Game Loop Instrumentation

To capture the random idle GPU cycles from games, we need to monitor the frame execution information, i.e., the start and end rendering time of each frame, in real time. Nowadays, there are many popular frame monitoring software for rendering workloads including PresentMon [17], IntelGPA [9], GpuView [8], and FrameView [5]. They all use event-tracing technology [4], which records events with high latency (usually >1 second). However, cloud gaming usually requires 60 frames per second, i.e., 16.67 ms per frame, which cannot accept such a large tracing latency.

In PilotFish, we exploit the fact that most games are developed on common graphic libraries (e.g., DirectX [3],

Table 2: The Game Loop Detector Performance.

	Avg. Overhead / 60 frame	Avg. Err.
ACOdyssey	0.1058 ms	0.363%
Genshin Impact	0.070668 ms	0.526%

Vulkan [20]), which translate the graphical operations into GPU commands. When the game finishes generating the draw calls, the GPU commands will be submitted to the GPU via a specific API (e.g., ExecuteCommandList in DirectX). PilotFish instruments the command submission API of graphic libraries to detect the start time of frame rendering. The instrumentation latency is very low, usually within 1 microsecond per frame. Moreover, to obtain when the rendering completion time of the submitted frame, PilotFish inserts an additional GPU command for notification of rendering completion at the end of the submission queue. Since the QoS of cloud gaming determines the maximum frame rate, e.g., an FPS of 60 means there is at most one frame per 16.67ms. When PilotFish is notified with the rendering completion, we can calculate when the next frame would appear, thus the time period before the next frame is guaranteed to be idle. Table 2 shows the average overhead and error for FPS perception through the game loop detector. The overhead is negligible where the average overhead per 60 frames is around 0.1ms. We also validate FPS measured from PilotFish by comparing with PresentMon [17] as the ground-truth. The average measurement error of FPS is 0.526%. Instrumenting the graphic libraries that most games built on allows PilotFish to generally support a wide range of existing games and future games without specific modification for every game.

Algorithm 1: DL training scheduler

```
1 while true do
2   if isFrameRendering() then
3     WaitForFrameComplete();
4     freeTimeslice = FrameTimeQoS -
       LastFrameRenderingTime;
5   else
6     kernel = GetKernelFramePool();
7     kernelTime = PredictDuration(kernel);
8     if freeTimeslice > kernelTime then
9       LaunchKernel(kernel);
10      freeTimeslice =
        freeTimeslice - kernelTime;
```

5 DL Training Scheduler

With the captured GPU idle periods, PilotFish will schedule the computation kernels from DL training to harvest the free GPU cycles. As shown in Figure 9, PilotFish only allows the DL kernels to execute within the idle GPU periods to avoid GPU contention. Algorithm 1 describes the scheduling strategy of PilotFish: (1) when the game is using GPU to perform rendering, it will wait for the notification of the rendering completion; (2) when the game finishes rendering a frame, the scheduler sends the DL kernels that can finish before the deadline when FPS QoS is affected (e.g., when the QoS is 60 frames per second, the start time between two frames should be no more than 16.67 ms).

PilotFish’s DL training scheduler relies on the prediction of computation kernels to decide whether the submitted kernel can finish before the next frame starts (Line 7 in Algorithm 1). PilotFish leverages the predictability and iterative pattern of DL training. The kernels for the same model will be repeatedly submitted in every iteration with different input data. As we have shown in Figure 6, the kernel duration has a very low variance, which can be easily obtained via offline profiling. In PilotFish, the DL training jobs to co-locate with games will be profiled on idle GPUs for tens of iterations (usually a few minutes), and record their kernel execution time.

Note that, the GPU context for DL training is first created in the job initialization, thus its overhead does not affect the scheduling of DL kernels. Also, the launching of computation kernels has an overhead of 10 us, which is usually less or equal to a kernel’s execution time. To hide the kernel launching overhead, like most training frameworks, PilotFish submits the computation kernels asynchronously (as shown in Figure 9). Therefore, PilotFish only suffers from at most one kernel launching overhead at the first DL kernel in each frame, which is negligible.

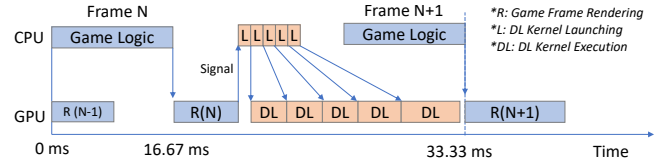


Figure 9: Fine-grained scheduling of DL kernel.

6 Task Executor

After the DL kernel is scheduled, the task executor monitors the kernel execution to avoid straggler kernels that run longer than expected and do not finish before the next rendering frame. In case the potential interference could appear, the task executor will terminate the process quickly to reclaim the GPU for game rendering while minimizing the loss of training progress. In addition to GPUs, the task executor also manages the other resource types including CPUs, PCIe bus and Disk I/O to avoid non-GPU interference.

6.1 GPU Kernel Execution

During the execution of computation kernels, PilotFish’s task executor keeps monitoring the running kernels on their execution time. Although, not often, some straggler kernels may run longer than the predicted time, which may postpone the rendering if they do not finish before the next frame appears. Note that the straggler kernels will not lead to QoS violation if the rendering time of the next frame is short and can still be finished within the deadline. Also, a slight drop in FPS (1 ~ 2 FPS) may not affect the gaming experience for non-sensitive players. Therefore, PilotFish provides two types of guarantees:

1) Hard guarantee: once a straggler kernel appears that it can not finish before the next frame rendering begins, the task executor suspends the running DL kernel on the GPU immediately.

2) Soft guarantee: PilotFish does not terminate the straggler kernels unless FPS drop exceeds a certain threshold.

Using soft guarantee is more friendly to DL training models that contain kernels of long execution time, e.g., the longest kernel of LSTM runs for 2.4 ms. Our evaluation in § 7.4 shows using the soft guarantee can harvest over 30% more GPU cycles than the hard guarantee when we co-locate LSTM with RDR2.

6.2 Low-overhead Pause and Resume

Figure 10 shows the design of PilotFish’s DL training pause and resume. In order to terminate the straggler kernel quickly, PilotFish leverages the multi-priority streams of modern GPUs to send asserting signal to DL training kernels at the highest priority. The preemption can be done very fast within 0.7 ms. However, asserting the kernel would wipe out all the

memory state that results in loss of the training progress. Although DL training may periodically save checkpoints, it is done in a less frequent manner (usually every a few epochs that takes hours).

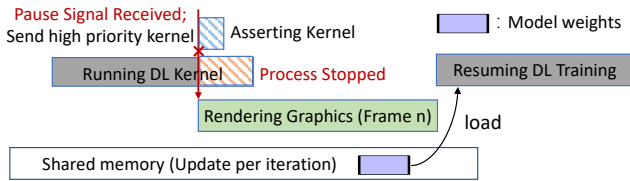


Figure 10: PilotFish’s low-overhead pause and resume.

Note that, we only want to terminate the computation to avoid the interference to games thus it is not necessary to clear the memory. To maintain the model weight while DL training job suspension, PilotFish builds a shared memory pool in an isolated process, that stores a backup version of the model weight. When resuming a DL training job from suspension, the pointer of shared memory is directly shipped to the memory manager of the training frame. If the GPU supports inter-process communication (IPC), the shared memory pool is placed on GPU thus no memory copy is needed. Otherwise, The shared memory pool is placed on the host memory thus requires resume the model weights by copying them from host to GPU. Our evaluation shows resuming the model from the host memory for ResNet-34, VGG-16, MobileNet and LSTM takes 64, 69, 63 and 30 ms respectively. But PyTorch’s requires over 7 seconds via its default checkpointing mechanism.

6.3 Mitigating Other Resource Contention

In addition to contention on GPU, both cloud gaming and the DL training involve other resource types thus also need careful management for interference avoidance.

CPU contention. For the DL training tasks, the CPU is used for data pre-processing, e.g., image decoding, re-shaping, data augmentation. Games use CPU for processing game logic and simulate physical effects. CPU contention may appear when the CPU-heavy DL training and games are co-located, resulting in a decrease in FPS and an increase of game loading time. PilotFish solves the resource contention on CPU by setting the priority of threads: game threads use a high priority and DL training threads use a low priority. Figure 11 shows the FPS of RDR2 to be co-located with a job that only pre-processes the data of DL training. By increasing the stress of the co-located job, the FPS and loading time of the game is affected severely if they have the same CPU thread priority. The Windows OS’s scheduler can fully mitigate the interference on CPU after we set the thread priority of the co-located job to low.

PCIe contention. Using PCIe bus, games transfer vertex data and primitive data from pageable memory to GPU dur-

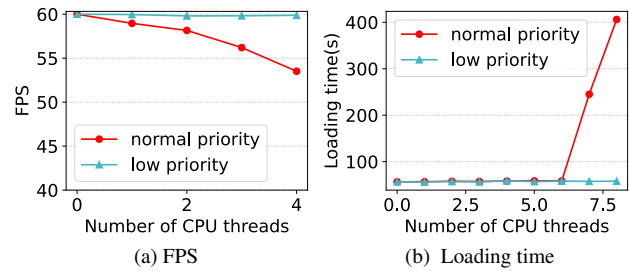


Figure 11: The FPS and loading time of RDR2 when co-located with CPU threads for DL training.

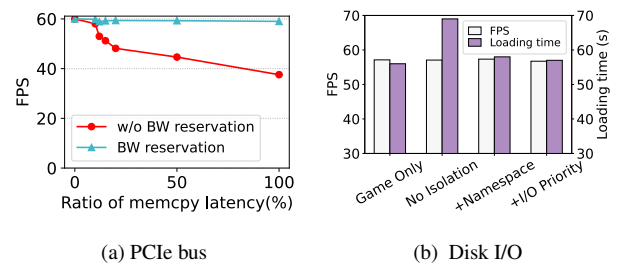


Figure 12: The inference to RDR2 on PCIe bus and Disk I/O.

ing execution, and the rendered frames are passed back from GPU [21]. The DL training uses PCIe to transfer data and model parameters. We have tested two popular games’ performance benchmarks (Shadow of the Tomb Raider and The Division 2). The average memcopy time per frame is 0.1748ms and the frames with copy time greater than 0.5ms account for 3.9% of all frames. When games use pageable memory and transfer data through PCIe bus alone, the achieved data transfer rate is 11,045MB/s. Because the theoretical peak bandwidth of 16x PCIe 3.0 bus used in our platform is 15,800MB/s and the effective bandwidth is 12,160MB/s, the bus can only support at most $\lfloor \frac{12160}{11045} \rfloor = 1$ memcopy task to transfer data in their full speeds in the same direction. Therefore, it is necessary to guarantee no interference on PCIe to avoid the game’s data transfer. In PilotFish, we rely on the bandwidth reservation technique proposed in Baymax [25] to reserve the enough PCIe bandwidth for cloud gaming. The DL training can only transfer data when the game is not using PCIe.

Figure 12a shows the FPS of game RDR2 when co-located with a stress test progress of memory copy. This stress test copies data from the host memory to the global memory of GPU, and then back to the host memory every 60 ms. We control the proportion of the memcopy time to the total time by controlling the size of the copied data. With increased memory copy stress, the FPS drops greatly without reserving the PCIe bandwidth for the game. The reservation guarantees the game is not affected by PCIe contention.

Disk I/O contention. From disk, games loads rendering

resources (e.g., texture) and DL training loads training data. Contention on disk I/O may lead to longer loading time for games. Figure 12b illustrates the FPS and loading time of a game co-located with a disk stress benchmark performing sequential read/write and 4K read/write [2]. Without any isolation, the FPS does not change but the loading time is increased by 21%. Moreover, we observe some objects are not rendered in the displayed frame, which is unacceptable to players. We apply the widely used I/O isolation techniques, including namespace [12] and I/O priority [10]. We find both techniques can guarantee the game performance by isolating the I/O operations.

GPU memory and caches. To avoid swapping data among GPU memory and host memory, PilotFish only co-locates a game and a DL training job when the sum of their peak GPU memory demand can fit into the GPU memory. Since DL kernels are only executed in the idle GPU cycles, their data movement between GPU memory and GPU caches has no overlap with gaming. GPU commands from game and DL training are serialized without preemption thus there is no context switching overhead. DL training may flush the GPU cache of rendering data of the previous frame. But we do not observe impact on rendering time to the next frame.

Network and video stream encoding. In PilotFish, we assume the distributed training uses a separate network from the cloud gaming service due to security and performance concern, thus there is no interference in network. Also, as we have explained in Figure 2, video stream encoding is done in a separate hardware encoder, thus is not interfered by DL training.

7 Evaluation of PilotFish

We have implemented a prototype of PilotFish on DirectX 12, CUDA 11.1, Windows 10, and PyTorch 1.8 with 2400 lines of code. As far as we know, PilotFish is the first system that co-locate cloud gaming with DL training. Therefore, we compare PilotFish with several straw-man solutions to evaluate its effectiveness. Overall, PilotFish can harvest up to 85% of idle GPU cycles from cloud gaming without generating interference.

7.1 Experimental Setup

We evaluate PilotFish with Steam Remote Play & Steam Link (cloud gaming platform) using the Nvidia RTX 2060 GPU. Table 3 summarizes the software and hardware experimental configurations. Note that PilotFish does not rely on any special hardware features of RTX 2060, and is easy to be set up on other GPUs. As listed in Table 4, we use five popular DirectX 12 games and four DL training applications to perform the experiment.

Throughout our experiments, the FPS target of games is 60 FPS (16.67 ms/frame). The QoS of the game is defined

Table 3: Hardware and software specifications.

	Specification
Hardware	Intel(R) i7-7700 @ 3.60GHz Nvidia GeForce RTX 2060
Software	Windows10 19043.1110 CUDA Driver 11.1.96 CUDA SDK 11.1 DirectX 12.1 PyTorch 1.8.1

Table 4: Benchmarks used in the experiment.

Benchmarks	Workloads
Ashes of the Singularity (AOS)	Crazy quality on 2560*1440; FPS: 60 GPU focused benchmark
Red Dead Redemption 2 (RDR2)	Favor performance quality on 2560*1440; FPS: 60
Shadow of the Tomb Raider (SOTTR)	High quality on 2560*1440; FPS: 60
F1 2021 (F1)	Medium quality on 1920*1080; FPS: 60
HITMAN3 (HIT3)	Ultra quality on 2560*1440; FPS: 60
DL Training	ResNet-34 (RS) [30]; VGG-16 [44]; MobileNet (MN) [31]; LSTM [45]; Dataset: ImageNet-1k, Wikitext-2

as the 99%-ile latency normalized to 60 FPS. We calculate the GPU utilization as the portion of time when the GPU is busy, which is the same as the definition of nvidia-smi [16]. We define the metric, harvest ratio, as the portion of GPU idle time that is harvested for DL training, which is calculated as

$$\text{Harvest Ratio} = \frac{\text{GPUUtil}_{\text{co}} - \text{GPUUtil}_{\text{Game}}}{100\% - \text{GPUUtil}_{\text{Game}}}, \quad (1)$$

where $\text{GPUUtil}_{\text{Game}}$ is the GPU utilization of running game independently, and $\text{GPUUtil}_{\text{co}}$ is the GPU utilization when game and DL training are co-located. For PilotFish, the time of model checkpointing is not considered as harvested.

Comparison Baselines. To compare the performance of PilotFish, we propose three straw-man solutions:

1. **GameMode** [6] is a feature introduced by Windows to prioritize CPU threads of games. It does not control GPU execution.
2. **Constant-Speed** controls the DL kernel submission speed with a constant rate.
3. **Adaptive-Speed** controls the DL kernel submission speed dynamically according to the FPS profiled from the event-tracing tool PresentMon [17]. If $\text{FPS} < 60$, the DL kernel submission speed is halved, otherwise, it is multiplied by 1.2.

7.2 GPU Utilization Improvement and FPS Guarantee

We first demonstrate the effectiveness of PilotFish by comparing PilotFish with the three baselines on all combinations of cloud games and DL models listed in Table 4. By default, the Constant-Speed baseline is set to 50% of the ideal speed (i.e., training the model on the same GPU without co-location).

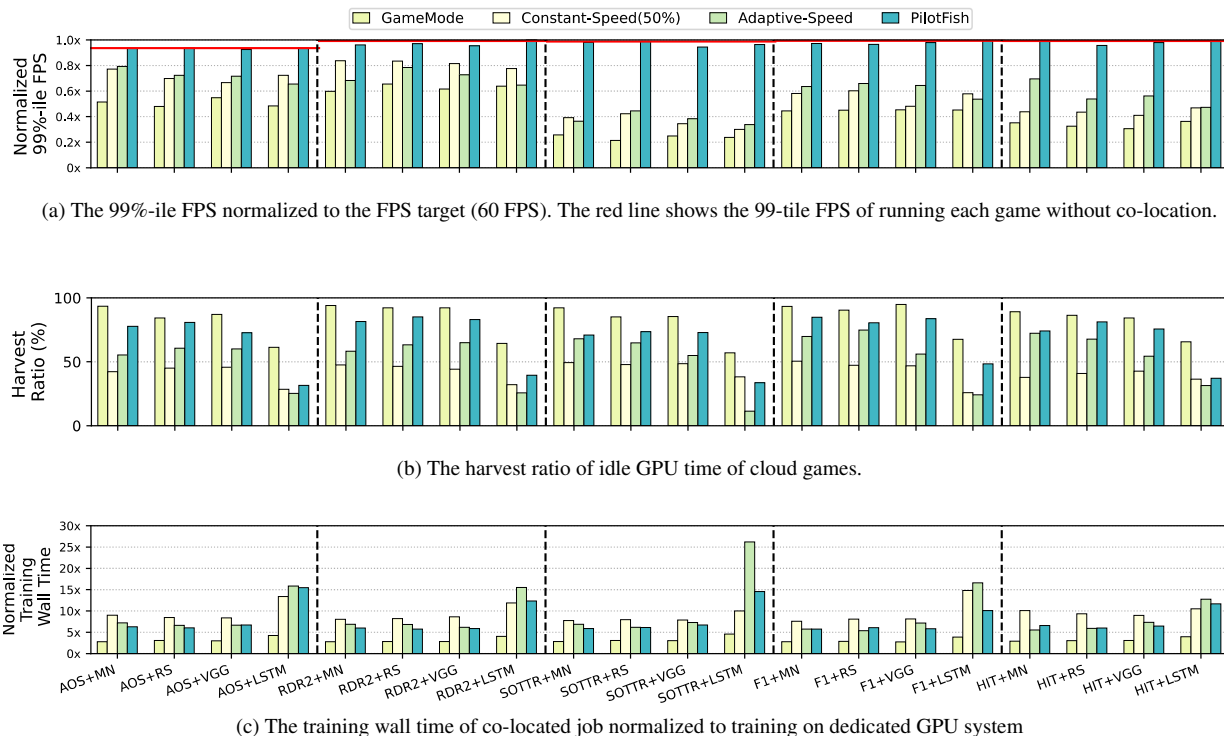


Figure 13: The 99%-ile FPS, harvest ratio and training wall time of different co-location combinations of games and DL models.

Figure 13 presents the 99%-ile FPS of the cloud games normalized to the FPS target (60 FPS), and the harvest ratio of idle GPU time. Note that, due to bursty complex frames, the cloud game may not always maintain at 60 FPS either even without co-location. Figure 13a shows that PilotFish achieves almost the same 99%-ile FPS compared to that without co-location. The three baselines all experienced severe FPS drops. GameMode drops the most, by up to 78.6% (e.g. SOTTR+RS). Constant-Speed(50%) and FPS-Based drop from 16.3% to 69.2% and from 20.7% to 66.3%, respectively. In the game of SOTTR and HIT, all baselines suffer from severe interference. Since SOTTR switches scenes multiple times during the benchmark, its rendering time of frames fluctuates more severely than the other four games. The three baselines cannot quickly adapt to the fluctuation thus perform poorly.

Figure 13b shows the harvest ratio in the different combinations. As shown in the figure, PilotFish w/ hard guarantee harvests 78.56% of idle times on average in all five games co-located with three DL training tasks (MobileNet, Resnet-34 and VGG-16) without interfering with the cloud games. When cloud games are co-located with LSTM, the harvest ratio drops to 39.03%. Because LSTM contains some large kernels that run for ~ 2.4 ms, they may not be scheduled if the idle GPU time is short with PilotFish’s hard guarantee. Since the rendering time of game F1 is lower than other games, its harvest ratio on LSTM is relatively higher than others, which is 48.43%. With the huge penalty of FPS drop, GameMode

achieves the highest harvest ratio (83% on average) since it does not control the speed of DL training. The harvest ratios of the Constant-Speed (50%) and Adaptive-Speed range from 26% to 50% and 11% to 74%. These two baselines not only harvest less idle GPU time than PilotFish but also degrades the FPS significantly. They prove the necessity of PilotFish’s mechanisms to fast and safely schedule DL kernels.

Figure 13c shows the training wall time of the co-located DL models normalized to training them on dedicated GPU. The training wall time is almost inversely proportional to the harvest ratio. Because GameMode occupies more GPU cycles from games in addition to the idle cycles, it has the least slowdown at the price of severely affected game FPS. Because of higher harvesting efficiency, PilotFish’s training wall time is better than Constant-Speed and Adaptive-Speed for most models without affecting the FPS of games.

7.3 Dissecting Execution

To demonstrate how cloud game runs when co-located with DL training, in Figure 14, we show the instantaneous FPS (the inverse of frame time) fluctuation of RDR2 over time when co-located with ResNet-34. We select a game segment (50 seconds) during the stable running of the game. We find PilotFish can always be stable near the original FPS without co-location. The baselines experience serious FPS fluctuations, especially GameMode and Constant-Speed since they are not

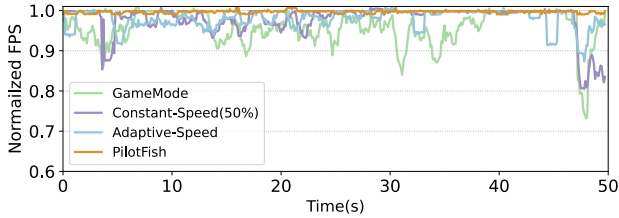


Figure 14: The instantaneous FPS of RDR2 over time when co-located with ResNet-34. The FPS is normalized to the average FPS without co-location.



Figure 15: The rendering quality of PilotFish (left), No co-location (middle), and GameMode (right). The rendering quality in GameMode is much worse than the others due to interference.

adaptive at all. The fluctuated and degraded FPS leads to very poor experience for players. When FPS drops, some games with adaptive rendering mechanism will actively reduce the rendering quality to maintain a smooth playing experience. Figure 15 compares the rendering quality of PilotFish, no co-location, and GameMode. PilotFish has the same rendering quality with running the game without co-location. But the game co-located with DL training in GameMode reduces the rendering quality under the bridge due to interference.

7.4 Sources of Improvement

Dynamic Scheduling. Figure 16 shows normalized 99%-ile FPS and the harvest ratio of the pair (RDR2+RS) under the different kernel submission speed in Constant-Speed policy. The kernel submission speed ranges from 3% to 100% (normalized to the ideal speed without co-location). The right-most column shows the results of PilotFish for comparison. As expected, we find that the 99%-ile FPS decreases and the harvest ratio increases as the submission speed grows from 3% to 100%. We specifically listed the 99%-ile FPS at the kernel submission speed 3% and 4%. We find the FPS target can be satisfied only when the kernel submission speed is very low. When the submission speed is higher than 4%, 99%-ile FPS begins to drop. Without degrading the 99%-ile FPS, PilotFish can achieve the same harvest ratio of Constant-Speed at 80% submission speed.

Figure 17 shows the harvest ratio of HIT under different rendering qualities when co-located with MobileNet and LSTM.

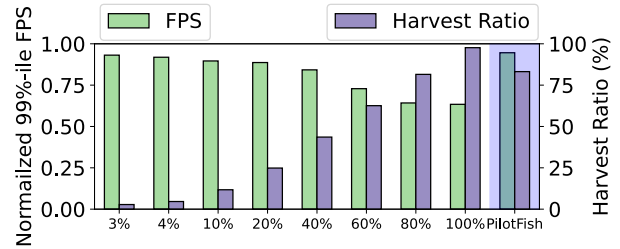


Figure 16: The impact of kernel submission speed in Constant-Speed v.s. PilotFish.

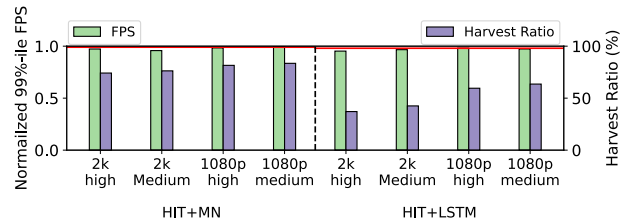


Figure 17: (HIT+MN/LSTM) the 99%-ile FPS and harvest ratio of PilotFish with different graphic quality.

Since MobileNet is mainly comprised of small kernels, it is easier to fit into short idle GPU periods. But the LSTM model contains some long kernels (~ 2.4 ms), which requires longer idle GPU periods. Therefore, reducing the rendering quality allows LSTM to harvest more GPU cycles from the game than MobileNet.

The two experiments in Figure 16 and Figure 17 imply the dynamic scheduling of DL kernels is necessary to handle the high randomness of game frames and diverse characteristic of different combinations of game and DL model.

Effective training pause and resume. To verify the need for the training pause mechanism introduced in Section 6.1, we disable this mechanism to compare its impact with PilotFish's policies of hard guarantee and soft guarantee. Figure 18 shows the 99%-ile FPS (normalized to 60 FPS) and the harvest ratio with different pause policies. When using the policy of hard guarantee, PilotFish achieves the same FPS of that without co-location. When the pause condition is relaxed by 5% (3 FPS), the FPS using the policy of soft guarantee is degraded within the threshold while the harvest ratio is increased. When disabling the pause mechanism, the FPS further decreases at the cost of no FPS guarantee. The impact of the pause policies is different for models: ResNet-34 is less impacted than LSTM since the computation kernels of ResNet-34 is much shorter than LSTM. In the worst case, if a DL training task submits a long-running kernel, the game rendering could be infinitely postponed. Therefore, we suggest using the policy of soft guarantee when the cluster opera-

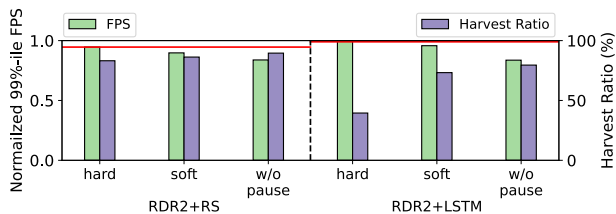


Figure 18: The 99%-ile FPS and harvest ratio of PilotFish with different training pause policies.

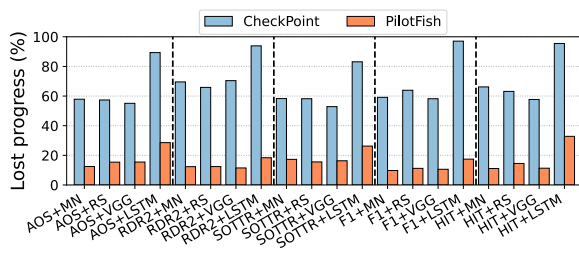


Figure 19: (RDR2+RS) the lost progress of DL training.

tor wants to trade a limited interference with a higher GPU utilization.

Figure 19 shows the lost progress of DL training due to training pause with the hard guarantee. Compared with the epoch-level checkpoint in PyTorch whose lost progress is 69.62% on average, PilotFish reduces the lost progress by 4.6 times to 15.04% with the weight backup in the shared memory pool. LSTM loses more training progress than other models since it triggers more training pause due to its longer computation kernel.

8 Scale to Data Center

To evaluate the potential benefit of PilotFish to cloud gaming service running in a large cluster of GPUs, we use a simple heuristic cluster-level scheduler to decide which games and DL training jobs are suitable for co-location on the same GPU, as shown in Figure 20. The heuristic cluster scheduler collects the average resource usage of DL jobs and games through offline profiling. It greedily matches the DL training job with the game with the DL training job so that the remaining resource is minimized. For the DL model using synchronous data parallel training, the scheduler prefers to deploy each of its workers to the servers with a similar utilization so that each worker can run at a similar speed, which can reduce the synchronization overhead.

We compare this heuristic policy with a random scheduling policy that co-locates games and DL training models randomly. We simulate a cloud gaming cluster composed of one thousands Nvidia RTX2060 GPUs. We select ten popular games as the workload of cloud gaming, including Dota 2,

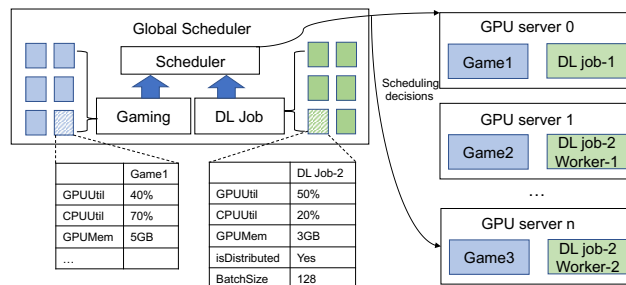


Figure 20: Cluster-level Scheduler.

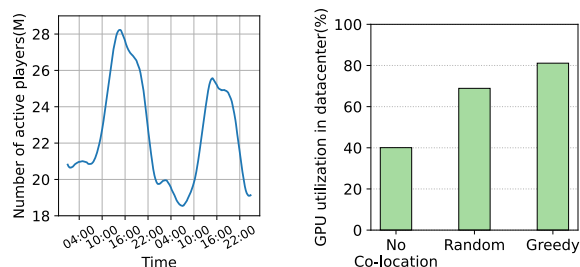


Figure 21: The variation of active players on Steam. Figure 22: The GPU utilization in the simulated cluster.

League of Legends, PUBG, CS:GO, Civilization 5, Assassin's Creed Odyssey, The Division 2, Ashes of the Singularity, RDR2, and Genshin Impact. The games are launched with the same probability. The number of running games follows the active player variation reported by Steam (shown in Figure 21), which has a strong diurnal pattern. We regard the peak point in Figure 21 as the situation when all the 1000 GPUs are used by cloud games. For the DL Training workloads, we select 750 instances evenly from the five models: ResNet-34, ResNet-50 [30], VGG-16 [44], MobileNet [31], and DenseNet [32]. Each model has 100 non-distributed training instances and 50 distributed training instances.

Figure 22 shows the GPU utilization of no co-location, random scheduler and the greedy heuristic scheduler. When there is no co-location, the cloud gaming cluster only has a GPU utilization of $\sim 40\%$. The random policy can improve the utilization to 68.89% due to PilotFish's efficient execution. Since the greedy policy is aware of the resource usage pattern of cloud games and DL training, it can further improve the cluster utilization to 81.12%. It implies the games and DL training jobs should be carefully scheduled at the cluster-level to maximize the benefit of co-location, which is an interesting future direction.

9 Related Work

General CPU co-location. There has been a large amount of prior work focusing on improving application QoS and

hardware utilization for CPU co-location. They can be broadly categorized as (1) profiling-based methods [26, 27, 41, 48, 52] and (2) partitioning-based methods [39, 53]. The profiling-based methods, such as Bubble-Up [41], Bubble-Flux [48], SMiTe [52], uses offline profiling of user-facing services and batch applications to predict their performance degradation to avoid contention on shared cache and memory bandwidth. They periodically adjust the allocation of shared resources according to the QoS feedback of user-facing services.

However, these techniques would fail on cloud gaming because they neglect the complex interaction of interference on different shared resources on GPUs.

General GPU co-location. Several techniques were proposed in the prior work to improve the utilization of GPUs with co-location. TimeGraph [34] and GPUSync [28] use priority-based scheduling to guarantee the performance of real-time kernels. High-priority kernels are executed first if multiple kernels are launched to the same GPU. GPU-EvR [35] launches different applications to different streaming multiprocessors (SMs) on one GPU. However, they are not applicable to our problem because they all rely on the simulator to synthesize the execution trace of co-located applications. Laius [50] and Baymax [25] predict the kernel duration and reorder the kernel based on the QoS headroom of user-facing queries. But it is difficult to predict the rendering frame time of game with a low overhead. AntMan [47] only co-locates multiple DL training jobs, which cannot handle the unpredictable game rendering. Nvidia Volta MPS (Multi-Process Service) [42] enables multiple applications to share a GPU concurrently with static partition, however, cannot handle the dynamic load of cloud gaming. Moreover, MPS-based solutions rely on the special hardware feature that only supports CUDA applications but not games, and is not applicable to non-Nvidia GPUs.

Co-location of cloud gaming. Specifically for cloud gaming, several works have been proposed to improve resource utilization by co-locating multiple games [37, 43, 49]. vGASA [49] adaptively schedules rendering tasks from multiple games to meet the SLA in a best-effort manner. However, when a hard SLA guarantee is required, vGASA has to reserve the resource for the worst cases so that all running games can meet the SLA at the most complex scenes. As we have shown in Section 2.2, cloud gaming has a high variance in GPU usage. Conservatively guaranteeing the worst case would waste resources with a significant over-provisioning.

GAugur [36] and dJay [29] dynamically tune the game settings for the co-located games during gameplay to adapt to changes of game scenes for improving performance. However, as we have shown in Figure 4, the frame time and GPU load in the gaming could fluctuate drastically even within a short period of time. Frequently changing the game setting is noticeable to players and could greatly degrade the gaming experience. This is unacceptable to commercial cloud gaming services. Instead, the computation of DL training is highly

predictable. PilotFish can accurately predict the execution of DL kernels and schedule them only when it is safe. This is the main reason why we claim DL training is the right workload to be co-located with cloud gaming.

10 Conclusion

Cloud gaming service suffers from low GPU utilization issue due to the limitation of network and edge devices. Since cloud gaming utilizes GPUs in a very random manner, existing co-location solutions for GPU cannot meet the QoS requirement of cloud gaming. PilotFish addresses this issue by co-design cloud gaming service and deep learning training framework. PilotFish can harvest free GPU cycles using DL training with no interference to cloud gaming. PilotFish achieves this hard guarantee by (1) quickly capturing the idle GPU periods from cloud gaming via low-overhead instrumentation to graphic libraries (e.g., DirectX); (2) leveraging the predictability of DL computation to safely schedule DL kernels; and (3) providing a low-overhead mechanism to pause DL computation when they could potentially interfere with games. Our evaluation shows that PilotFish can harvest a significant portion of idle GPU time of cloud gaming up to 85.1% without affecting the gaming experience. PilotFish reveals a principled design to co-locate unpredictable workloads with predictable low-priority workloads. In addition to co-locating cloud gaming with DL training, it is interesting to generalize PilotFish's solution on other predictable workloads, e.g., scientific computing [22, 50].

Acknowledgments

This work is partially sponsored by the National Natural Science Foundation of China (62022057, 61832006, 61872240), and Shanghai international science and technology collaboration project (21510713600). We thank the anonymous reviewers for their constructive feedback and suggestions. Zhenhua Han, Quan Chen and Minyi Guo are the corresponding authors.

References

- [1] Amazon appstream. <http://aws.amazon.com/appstream>.
- [2] As ssd benchmark. <https://www.alex-is.de/PHP/fusion/news.php>.
- [3] DirectX. <https://docs.microsoft.com/en-us/windows/win32/directx>.
- [4] Event tracing for windows. <https://docs.microsoft.com/en-us/windows/win32/etw/about-event-tracing>.

- [5] Frameview. <https://www.nvidia.com/en-us/geforce/technologies/frameview/>.
- [6] Gamemode. <https://support.xbox.com/en-US/help/games-apps/game-setup-and-play/use-game-mode-gaming-on-pc>.
- [7] Google stadia. <https://stadia.google.com>.
- [8] Gpuview. <https://docs.microsoft.com/en-us/windows-hardware/drivers/display/using-gpuview>.
- [9] Intelgpa. <https://software.intel.com/content/www/cn/zh/develop/tools/graphics-performance-analyzers.html>.
- [10] I/o prioritization in windows os. <https://clightning.medium.com/i-o-prioritization-in-windows-os-6a0637874a52>.
- [11] Microsoft xbox remote play. <https://www.xbox.com/en-US/consoles/remote-play>.
- [12] Namespace. <https://docs.microsoft.com/en-us/windows/win32/adsi/namespaces>.
- [13] Nvenc. <https://developer.nvidia.com/nvidia-video-codec-sdk>.
- [14] Nvidia cuda. <https://developer.nvidia.com/zh-cn/cuda-toolkit>.
- [15] Nvidia geforce now. <https://www.nvidia.com/en-us/geforce-now/>.
- [16] Nvidia system management interface. <https://developer.nvidia.com/nvidia-system-management-interface>.
- [17] Presentmon. <https://github.com/GameTechDev/PresentMon>.
- [18] Sony playstation now streaming. <http://us.playstation.com/playstationnow>.
- [19] Steam survey. <https://store.steampowered.com/stats/Steam-Game-and-Player-Statistics>.
- [20] Vulkan. <https://www.vulkan.org/>.
- [21] Wei Cai, Ryan Shea, Chun-Ying Huang, Kuan-Ta Chen, Jiangchuan Liu, Victor CM Leung, and Cheng-Hsin Hsu. A survey on cloud gaming: Future of computer games. *IEEE Access*, 4:7605–7620, 2016.
- [22] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.
- [23] Quan Chen, Zhenning Wang, Jingwen Leng, Chao Li, Wenli Zheng, and Minyi Guo. Avalon: towards qos awareness and improved utilization through multi-resource management in datacenters. In *Proceedings of the ACM International Conference on Supercomputing*, pages 272–283, 2019.
- [24] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 17–32, 2017.
- [25] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ACM SIGPLAN Notices*, 51(4):681–696, 2016.
- [26] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, volume 48, pages 77–88. ACM, 2013.
- [27] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.
- [28] Glenn A Elliott, Bryan C Ward, and James H Anderson. Gpusync: A framework for real-time gpu management. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 33–44. IEEE, 2013.
- [29] Sergey Grizan, David Chu, Alec Wolman, and Roger Wattenhofer. djay: Enabling high-density multi-tenancy for cloud gaming servers with dynamic cost-benefit gpu load balancing. In *Proceedings of the sixth ACM symposium on cloud computing*, pages 58–70, 2015.
- [30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [31] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.
- [32] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869*, 2014.

- [33] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant {GPU} clusters for {DNN} training workloads. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 947–960, 2019.
- [34] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011.
- [35] Haeseung Lee, Al Faruque, and Mohammad Abdullah. Gpu-evr: Run-time event based real-time scheduling framework on gpgpu platform. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 220. European Design and Automation Association, 2014.
- [36] Yusen Li, Chuxu Shan, Ruobing Chen, Xueyan Tang, Wentong Cai, Shanjiang Tang, Xiaoguang Liu, Gang Wang, Xiaoli Gong, and Ying Zhang. Gaugur: Quantifying performance interference of colocated games for improving resource utilization in cloud gaming. In *Proceedings of the 28th international symposium on high-performance parallel and distributed computing*, pages 231–242, 2019.
- [37] Yusen Li, Changjian Zhao, Xueyan Tang, Wentong Cai, Xiaoguang Liu, Gang Wang, and Xiaoli Gong. Towards minimizing resource usage with qos guarantee in cloud gaming. *IEEE Transactions on Parallel and Distributed Systems*, 32(2):426–440, 2020.
- [38] Tianyi Liu, Sen He, Sunzhou Huang, Danny Tsang, Lingjia Tang, Jason Mars, and Wei Wang. A benchmarking framework for interactive 3d applications in the cloud. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 881–894. IEEE, 2020.
- [39] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.
- [40] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 881–897, 2020.
- [41] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259. ACM, 2011.
- [42] NVIDIA. Sharing a gpu between mpi processes: multi-process service(mps). Oct. 2012.
- [43] Zhengwei Qi, Jianguo Yao, Chao Zhang, Miao Yu, Zhizhou Yang, and Haibing Guan. Vgris: Virtualized gpu resource isolation and scheduling in cloud gaming. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(2):1–25, 2014.
- [44] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [45] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.
- [46] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.
- [47] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020.
- [48] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 607–618. ACM, 2013.
- [49] Chao Zhang, Jianguo Yao, Zhengwei Qi, Miao Yu, and Haibing Guan. vgas: Adaptive scheduling algorithm of virtualized gpu resource in cloud gaming. *IEEE Transactions on Parallel and Distributed Systems*, 25(11):3036–3045, 2013.
- [50] Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Edward Mawhirter, Bo Wu, Chao Li, and Minyi Guo. Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in data-centers. In *Proceedings of the ACM International Conference on Supercomputing*, pages 58–68, 2019.

- [51] Wei Zhang, Kaihua Fu, Ningxin Zheng, Quan Chen, Chao Li, Wenli Zheng, and Minyi Guo. Charm: Collaborative host and accelerator resource management for gpu datacenters. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 307–315. IEEE, 2021.
- [52] Yunqi Zhang, Michael A Laurenzano, Jason Mars, and Lingjia Tang. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 406–418. IEEE, 2014.
- [53] Haishan Zhu and Mattan Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 33–47, 2016.



Privbox: Faster System Calls Through Sandboxed Privileged Execution

Dmitry Kuznetsov
Tel Aviv University

Adam Morrison
Tel Aviv University

Abstract

System calls are the main method for applications to request services from the operating system, but their invocation incurs considerable overhead, which has been aggravated by mitigation mechanisms for transient execution attacks. Proposed approaches for reducing system call overhead all break the semantic equivalence between system calls and regular function calls (e.g., by making system calls asynchronous), and so their adoption requires rearchitecting applications.

This paper proposes *Privbox*, a new approach for lightweight system calls that maintains the familiar synchronous, function-like system call model. Privbox allows an application to execute system call-intensive code in a *semi-privileged, sandboxed* execution mode, called a “privbox”. Semi-privileged execution is architecturally similar to the kernel’s privileged execution, which enables faster invocation of system calls, but the code is sandboxed to ensure that it cannot use its elevated privileges to compromise the system. We further propose *semi-privileged access prevention* (SPAP), a simple hardware architectural feature that alleviates much of Privbox’s instrumentation overhead.

We implement Privbox based on Linux and LLVM. Our evaluation on x86 (Intel Skylake) hardware shows that Privbox (1) speeds up system call invocation by 2.2×; (2) can increase throughput of I/O-threaded applications by up to 1.7×; and (3) can increase the throughput of real-world workloads such as Redis by up to 7.6% and 11%, without and with SPAP, respectively.

1 Introduction

System calls are the de-facto method for processes to request services from the operating system (OS), but they are orders of magnitude slower than a regular function call. Much of the overhead stems from switching the processor’s execution mode between unprivileged user-mode and privileged kernel execution on system call entry and exit [1]. User-to-kernel mode switches are further slowed down by the protection mechanisms [2, 3] recently added to mitigate transient execution vulnerabilities such as Meltdown [4] and Spectre [5].

Reducing system call overhead has attracted significant research attention over the years (e.g., [6, 1, 7, 8, 9, 10]), and the increased overhead imposed by the mitigations of transient execution vulnerabilities [10] underscores the importance of addressing the problem. Current approaches, however, *break*

the semantic equivalence between system calls and regular function calls. For instance, FlexSC [1] and *io_uring* [8] make system calls asynchronous; the *io_uring* model and similar models [6, 11, 12] also limit how system calls can be composed. Consequently, benefitting from these system call designs requires rearchitecting applications to use the new system call models.

In this work, we propose *Privbox*: a new approach for lightweight system calls that maintains the familiar user-space programming model of synchronous, function-like system calls. In our design, an application can demarcate system call-intensive code and have it execute in a *privbox*, in which system call invocation is cheap—e.g., 2.2× faster than a regular system call on an Intel Skylake CPU. An application can thus enjoy low-overhead system calls with an unchanged synchronous system call model and only minor source code modifications to demarcate privboxed code regions.

Privboxed code runs in a “semi-privileged” mode. Semi-privileged execution consists of the processor running in privileged mode with the kernel address space mapped, which reduces user/kernel system call transition time, similarly to kernel-mode Linux (KML) [7]. But unlike KML, semi-privileged privboxed code runs *sandboxed*, so that it has the same access as the regular, unprivileged code of its process—thus it cannot violate OS security.

Our sandbox design is inspired by the software fault isolation approach of NaCl [13, 14], which uses compile-time instrumentation to generate verifiably safe code. We adapt NaCl’s instrumentation approach to the circumstances and environment of semi-privileged privboxed execution. In our design, source code demarcated for privboxed execution is instrumented at compile-time to prevent it from reading/writing arbitrary kernel memory or jumping to arbitrary kernel code. The kernel verifies the correctness of the instrumentation before allowing the code to begin its privboxed execution. The privboxed code then runs natively, without any runtime environment, but under a custom page table configuration that blocks it from executing uninstrumented user-mode code.

Unfortunately, the sandbox’s instrumentation slows down execution of the privboxed code, which reduces the benefit from faster system calls. We identify instrumentation of memory operations (load and store instructions) as the main culprit. Motivated by this finding, we propose *SPAP* (semi-privileged access prevention), a simple hardware architectural modification that enables omitting load/store instrumentation

from privboxed code. With SPAP, privileged mode hardware execution blocks an instruction from reading/writing a kernel address if that instruction resides in a user-mode address (as privboxed instructions do). SPAP's check can be implemented analogously to how x86-64 implements its supervisor mode access prevention (SMAP) [15] feature, which blocks privileged mode execution from accessing user-mode addresses.

We implement Privbox for x86-64 by adding support for semi-privileged execution to Linux and the musl standard C library [16], as well as extending LLVM to support generation of instrumented (sandboxed) code. We evaluate Privbox in two contexts:

1. Applications that use patterns such as I/O threads and reactors [17]. These patterns are system call intensive with little user-space logic, and are therefore less impacted by instrumentation overhead. We find that such applications can gain over $1.7\times$ speedup, even without SPAP.
2. Applications that combine I/O and user-space logic. Specifically, we modify Redis [18], memcached [19], and SQLite [20] to use Privbox. For simplicity, we compile the entire application with instrumentation, which gives us a lower bound on Privbox's benefit. Using Privbox in these applications requires little effort (20–30 lines of code) and yields an improvement of up to 7.6% on today's hardware without SPAP and up to 11% in a configuration that approximates performance under SPAP.

Contributions We make the following contributions:

- **Privbox design (§ 3).** We design Privbox, a new systems programming mechanism for lightweight system. Privbox maintains the familiar user-space system call model and its adoption requires no application source code changes.
- **Implementation (§ 4).** We implement support for Privbox for x86-64 in Linux, the musl C library, and LLVM.
- **SPAP (§ 5).** We propose semi-privileged access prevention, a simple hardware modification that enables omitting load/store instrumentation from privboxed code.
- **Evaluation (§ 7).** We show that Privbox improves (1) system call overhead by $2.2\times$; (2) I/O thread execution time by over $1.7\times$; and (3) real-world workload throughput by as much as 7.6% without SPAP and 11% with SPAP.
- **Availability.** The Privbox implementation and benchmarks are available at <https://github.com/privbox>.

2 Background & motivation

Monolithic kernels like Linux rely on hardware *privilege modes* to enforce process isolation and to mediate I/O access to peripheral devices. The kernel offers a set of *system calls* for processes to request services requiring OS mediation, such as input/output (I/O) to devices, inter-process communication, and virtual address space modification. System calls

are semantically equivalent to function calls, but are orders of magnitude slower. This problem has spurred research on reducing their overhead. These proposals, however, break the semantic equivalence between system and function calls, so adopting them requires rearchitecting applications to use a new system call model. Our goal is thus to address the problem without changing the system call programming model.

Privilege modes The basic hardware mechanism used to implement the OS isolation model is the distinction between unprivileged and privileged processor execution modes. The kernel runs in *privileged* mode, which allows full access to the entire instruction set, including *privileged instructions* for, e.g., installing a page table or enabling/disabling interrupts. Applications run in *unprivileged* mode, which only allows execution of non-privileged instructions that cannot circumvent OS isolation. Implementation of the execution privilege modes differs between hardware architectures. In this paper, we focus on the x86-64 architecture. It defines four hardware privilege levels, also called *rings*, numbered 0 to 3 in decreasing order of privilege. Most OSes execute user applications in ring 3 and the kernel in ring 0. The CPU has a *code segment* (CS) register that (indirectly) defines the CPU's current ring, also called *current privilege level* (CPL).

System calls A *system call* is a mechanism for a controlled and safe transfer of execution from untrusted unprivileged code to privileged kernel code. On x86-64, system calls are implemented by the `syscall` instruction. This instruction elevates the CPU's privilege mode and transfers control to a pre-determined kernel memory address, called the system call entry point. The entry point code determines the kernel function to service the desired call by inspecting certain CPU registers (determined by a software, OS-specific convention), executes it, and finally executes a "return to user" instruction which lowers the CPU's privilege mode and transfers control back to the unprivileged code.

System call overhead A system call is semantically equivalent to a function call from an application's perspective. But it is orders of magnitude slower, as its invocation/return is a multi-step procedure in both hardware and software, in which hardware elevates/lowers its privilege level and saves/restores certain CPU state, and kernel code saves/restores remaining CPU state and determines and executes the system call code.

System call overhead is exacerbated by mitigations of hardware microarchitecture vulnerabilities, such as Meltdown [4] and Spectre [5]. These vulnerabilities involve malicious user code abusing microarchitectural state shared by the CPU's privilege modes to read memory that is not architecturally accessible to the attacking user code. Mitigation accordingly involves modifying the relevant CPU state on system call entry, which adds overhead. For instance, on affected hardware, Linux's system call entry code flushes the CPU indirect branch predictor's state [3] to block Spectre v2 attacks. In addition, Linux's page table isolation (PTI) [2] Meltdown mit-

igation switches page tables during system call entry, which is a costly operation that also implies a TLB flush on x86-64. Indeed, our evaluation (§ 7.1) finds that while a standard system call invocation entry/exit time is $28\times$ slower than a function call/return, PTI makes it $52\times$ slower. And while Linux’s software mitigations are not used on recent processors that mitigate the vulnerabilities in hardware, the hardware mitigation itself slows down the system call instruction [10].

Reducing system call overhead There are several proposals to reduce the overhead of system call entry/exit. They generally achieve this by compromising on the semantic equivalence between system and function calls.

Flexible system calls (FlexSC) [1] makes system calls asynchronous instead of synchronous. It offloads system call execution to a kernel “syscall thread” associated with the process, with which the process communicates over a shared-memory interface, thereby eliminating CPU cycles spent on system call entry/exit. However, FlexSC requires *rearchitecting applications to use an asynchronous programming style*.¹ It also increases CPU usage due to the added threads and polling of the shared-memory communication structures.

System call overhead can be reduced by batching. The *multi-call* approach invokes several system calls with one kernel entry/exit. Linux includes several multi-calls, such as `preadv`, which performs a sequence of seek-followed-by-read operations. Cassyopia [6] explores compiler optimizations to batch several system calls together. But since a multi-call specifies the participating calls up front, it *does not support arbitrary composition of system calls and user-space logic offered by the standard system call model*.

Recent Linux versions offer an `io_uring` [8] mechanism. `io_uring` allows submitting I/O requests through a memory interface, like FlexSC, but it does not support arbitrary system calls. Like multi-calls, multiple requests can be submitted to a submission queue, but the submitted operations can be of different kinds and interact with different file descriptors. Overall, `io_uring` can be viewed as a combination of FlexSC and batching specialized for I/O, and thus suffers from the same limitations as those approaches.

Kernel bypasses avoid system call overhead by doing away with system calls for device access. For instance, DPDK [9] and SPDK [21] allow applications to interact directly with networking and storage devices, respectively. But since the kernel no longer mediates device access, its standard interfaces such as sockets or files cannot be used, and user-space has to implement all the abstractions it requires.

Ward [10] targets overhead related to Spectre and Meltdown mitigations. It constructs process page tables which do contain mappings of kernel memory, but only memory that is safe to expose to that process. At best, Ward reduces system

¹FlexSC offers a threading library that makes its asynchronous system calls transparent to applications, but this library is relevant only for applications with many user-mode threads.

call overhead to that of the pre-Spectre/Meltdown baseline, which is still significantly slower than a function call.

BPF for Storage [22] is a recent approach for reducing I/O path overhead by leveraging Linux’s eBPF subsystem [23], which is an in-kernel virtual machine that can execute user-loaded bytecode programs. The idea in BPF for Storage is to use eBPF programs to bypass kernel layers and avoid system calls, e.g., by searching an on-disk B+tree inside the kernel instead of via multiple system calls. However, *eBPF is a severely limiting programming model*, as the bytecode must pass static verification [24] before it can be executed in the kernel, and verification considerations limit eBPF programs to be small and to have provably bounded memory and execution time.

3 Design

Privbox is a new execution model for system call intensive code. Privbox provides standard synchronous, function-like system calls but with significantly lower invocation cost. Using Privbox thus requires no rearchitecting of application source code, as opposed to, e.g., the asynchronous system calls provided by FlexSC or `io_uring` (see § 2). We describe Privbox in the context of Linux on x86-64 hardware, but its design can be extended to other monolithic operating systems and/or hardware architectures.

Privbox provides an interface for executing code sections (e.g., ELF objects) in a *semi-privileged* execution mode (§ 3.1); we refer to such code as being *privboxed*. In semi-privileged execution, the processor is in privileged mode and kernel memory is mapped, which enables fast system call execution (§ 3.2), but for security, the code runs sandboxed, so that it has the same access as the process that loads it (§ 3.3). This sandbox is enforced by compile-time instrumentation (verified by the kernel) and by virtual memory restrictions.

Privbox and eBPF (§ 2) share some similarity in that both offer safe execution of user-supplied code in privileged processor mode, but the designs have a fundamental difference. eBPF runs code *in kernel context*, invoked to handle certain events, and so eBPF programs must be verified to terminate and be provided with interfaces for kernel operations. In contrast, privboxed code is *conceptually process code, just with faster system calls*. It can access the process’ address space, be scheduled and context switched, and can only interact with the kernel via the system call interface—in particular, it invokes the kernel and not vice versa. Privbox is thus a general-purpose design, whereas eBPF requires customization for each new use case (e.g., [22]).

While we focus on the general use case, in which privboxed code must be isolated from other processes in the system, some scenarios can use Privbox without sandboxing. One such example is a workload running by itself on a dedicated virtual machine. This use case can employ Privbox without sandboxing, as the privboxed code cannot compromise any-

```

main:
  fd = open("priv.so")
  call priv_code_load(fd)
  call priv_code_invoke("privfunc1")

privfunc1:
  loop:
    fast_syscall ...
  call priv_code_return()

```

Listing 1: Example usage of privcall mechanism.

thing other than itself.

3.1 Execution & usage model

Privbox introduces an operating system interface with the following system calls:

- *Load privboxed code.* This method can accept a pointer to a buffer with machine instructions or a file descriptor of an ELF object file. Internally, it verifies the code’s safety (§ 3.3) and relocates it to a dedicated, immutable memory region from which it will be executed.
- *Invoke privboxed code.* This method accepts a pointer or symbol from the loaded ELF file, and begins executing it in semi-privileged mode. I.e., from the calling application’s perspective, this method returns only when the privboxed code returns, as explained next.
- *Return from privboxed code.* This method can only be called by privboxed code. It exits semi-privileged execution and transfers control to the code that invoked it, while making a return value available.

Listing 1 shows an example program utilizing the above interface to invoke a function from the ELF object `priv.so`.

Usage model While an entire application can be privboxed, Privbox’s sandboxing instrumentation imposes overhead, so only code sections with a large fraction of cycles spent on system call entry/exit will benefit from privboxing. Privbox adoption thus consists of (1) the developer identifying system call intensive code sections for privboxing; (2) isolating such code sections into separate build units, which are built with the required instrumentation (e.g. into ELF objects); and (3) modifying application source code to load and invoke these objects as privboxed code at run time. Crucially, a privboxed code section itself requires no modification: it is simply another object file linked or loaded into the application.

We envision steps (2)–(3) being performed by tools, after developers demarcate privboxed code sections in the source code. In this paper, however, we perform them manually. For manual Privbox adoption, the “low hanging fruits” consist of applications whose software architecture already separates system call intensive code from computation code into distinct modules that communicate via some mechanism (e.g., SEDA [25]). Figure 1 depicts such an architecture. Because these architectures already isolate I/O (or other system call-heavy) code from other parts of the code, it is straightforward to surgically apply instrumentation only to that code. This approach minimizes Privbox’s instrumentation overhead, as compute heavy parts remain unaffected, while the I/O parts waste less cycles on system call entry/exit.

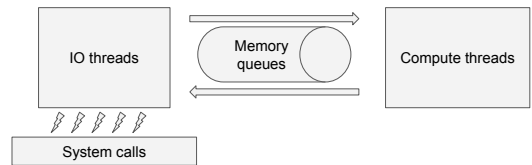


Figure 1: Software architecture that separates threads performing I/O (or system calls) from threads that perform other kinds of logic.

3.2 Semi-privileged execution

Privboxed code runs with the processor in privileged mode and kernel memory mapped, which enables it to perform a system call with a function call, without the `syscall` instruction, as detailed below. Except for having the ability for fast system calls, the OS treats privboxed code as user-space (process) code and its access is similarly restricted, hence the term *semi-privileged execution*. When privboxed code is invoked, the kernel transfers control to it with a new (ring 0) code segment (CS). Other than having a different CS value, the privboxed code runs similarly to unprivileged code—with interrupts enabled and the same priority, capabilities, and permissions. The kernel can preempt its execution at any moment and re-schedule it, as it would any other process.

Privboxed code runs with a custom page table, which modifies the standard virtual address space layout in several ways. Figure 2 shows the baseline layout of user and kernel memory in Linux. Privbox adds a special *privboxed code region* to the user part of the address space. This region is located in the lower part of the process address space and is unwritable by the process. After the kernel successfully verifies privboxed code, it relocates the code to the privboxed code region, from which it is later executed.

Normally, kernel mode execution has both kernel and user addresses mapped and accessible, and user mode execution only has user addresses accessible.² Privbox’s custom page table maps the same memory regions as the process’ page table, but using different access modes: user memory (excluding the privboxed code region) is marked not executable (see § 3.3) and kernel memory is marked accessible and executable, unlike user mode execution. (Despite kernel memory being

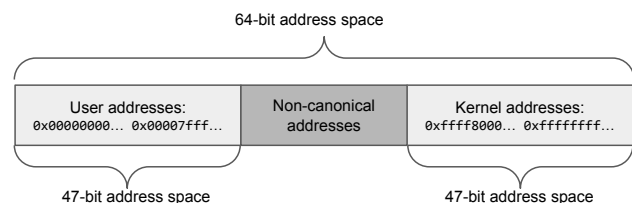


Figure 2: Example of memory map on x86-64 with 48-bit virtual address space. Lower address spaces are reserved for user program memory, higher address space are reserved for kernel memory. Access to memory at non-canonical addresses generates a trap.

²Without PTI, kernel addresses are still mapped, but their page table entries have a “supervisor” bit set, which allows access to the pages only when $CPL < 3$ (i.e., privileged mode). With PTI, kernel addresses are not mapped at all.

mapped accessible, privboxed code cannot directly access it due to the sandboxing instrumentation (§ 3.3.) Privbox’s custom page table is implemented by switching the top-level page table node to one in which the page entries (PTEs) mapping the user-space and kernel address ranges have the appropriate permissions, i.e., Privbox does not create nor maintain a copy of the process’ entire page table.

The execution environment of privboxed code enables a fast implementation of system calls. Privboxed code does not invoke a system call using the `syscall` instruction; it invokes a system call using a standard function call. Privbox provides a new kernel function that serves as a *gate* for system call invocation. This function has similar semantics to system call entry code: it sets up kernel execution environment (switching to kernel stack, storing registers) and routes requests to system call functions based on the system call number provided by calling code. Crucially, calls to this special function are not blocked by the sandbox instrumentation, so privboxed code can branch to it to perform a system call.

Modern x86-64 processors support supervisor mode access/execution prevention (SMAP [15]/SMEP), which disallow privileged execution from accessing/executing pages that are marked as user-space pages in the page table. Because privboxed code runs in “supervisor mode” but needs to execute and access user memory, SMAP/SMEP is disabled during privboxed execution. When privboxed code enters the kernel (e.g., a system call or interrupt), SMAP is re-enabled (this costs ≈ 26 cycles) but SMEP remains disabled (because toggling it costs thousands of cycles). § 6 discusses the security implications of this limitation.

A process that starts semi-privileged execution stays in that mode until it either invokes the “return from privboxed code” system call or an OS event that would result in non-sandboxed code execution occurs, such as invocation of an `exec` system call to load a new binary into the process address space. In such cases, the OS terminates the process’ semi-privileged execution, moving it to standard user-mode execution.

3.3 Sandboxing semi-privileged execution

Privbox must guarantee that a semi-privileged execution (of privboxed code) cannot perform any operation or memory access that the regular unprivileged execution of its process cannot perform. Importantly, this guarantee must also hold for transient execution of privboxed code (e.g., due to indirect branch misprediction) [4, 5]. This section describes our design for enforcing this safety property, which combines run-time virtual memory restrictions and compile-time instrumentation whose safety is verified by the kernel. § 6 analyzes the security of our design.

3.3.1 Sandboxing techniques

Three types of machine instructions can violate our desired safety property: (1) memory loads and stores, which have

Execution mode: \Rightarrow	User mode (CPL=3)	Privbox mode (CPL=0)	Kernel mode (CPL=0)
Accessed memory \Downarrow			
Kernel memory	N^\dagger/N^\ddagger	N^\ddagger/N^\ddagger	Y/Y
Privbox code	Y/Y	Y/Y	Y^*/Y^*
User memory	Y/Y	N^\dagger/Y	Y^*/Y^*

\dagger Restricted through page table access controls.

\ddagger Restricted through instrumentation.

* Subject to SMAP/SMEP.

Table 1: Access to memory regions under different execution modes. Left symbol: instruction fetching, right symbol: data load/store.

potential to access kernel memory; (2) control flow instructions, which can be used to branch into non-instrumented code; and (3) privileged instructions, usually reserved to operating system code. In the following, we describe how Privbox mitigates each of these risks.

Memory access and control-flow Privbox uses a combination of compile-time instrumentation and virtual memory (page table) protection to protect memory accesses. Table 1 summarizes which types of memory (rows) each type of execution mode (column) is allowed to access (for code execution/data), and how these restrictions are enforced.

We prevent privboxed code from executing user-space code outside of the (verified) privboxed code region, taking advantage of the “NX bit” feature of x86-64. On x86-64, each page table entry (PTE) has a No-Execute (NX) bit. When the NX bit is set, fetching instructions (code execution) is not permitted from the page(s) mapped by the PTE. Thus, the custom page table installed for Privbox’s semi-privileged execution maps user-space addresses outside the privboxed code region as non-executable.

We use compile-time instrumentation to ensure all other types of memory accesses are safe. Indirect load/store instructions (where the operand is known only at run time) are instrumented by introducing instructions that “sanitize” the memory operand, ensuring it does not point to kernel memory. For Linux on x86-64, we use the fact that clearing the most significant bit of a virtual address is guaranteed to create either a user address or an illegal non-canonical address, whose access will generate an exception (Figure 2).

Indirect control-flow instructions, such as calls, jumps, and returns are similarly instrumented, to ensure that privboxed code does not branch to arbitrary kernel addresses, because the kernel’s address space is mapped as executable. Our control-flow instrumentation also ensures that control-flow instructions can only branch to addresses which are verified when the code is loaded. Therefore, at run time, privboxed code can execute only instructions that were checked by the verifier.

Overall, our instrumentation approach guarantees that instruction memory operands are never kernel addresses, even for transient instructions, and thus blocks both non-transient and transient execution attacks (see § 6).

Privileged instructions We rely on Privbox’s verifier to check that loaded code does not contain privileged instruc-

tions. Rejecting code with privileged instructions does not limit Privbox’s applicability, as compilers do not emit such instructions unless specifically instructed.

3.3.2 Verification

The kernel verifies that privboxed code is correctly instrumented before allowing it to execute. Verification faces a challenge due to x86-64’s variable-length instructions, which mean that decoding the same code at different offsets can yield completely different instruction sequences. It is therefore possible that a seemingly benign instruction sequence would include malicious code at certain offsets [13].

To address this problem, our design relies on code being packed into *chunks*, which enables sound disassembly and verification of any possible execution of the loaded code. We define a code *chunk* as an aligned and fixed-length byte sequence containing machine instructions that always executes from its first to last instruction (i.e., a small fixed-size basic block). For example, 32-byte code chunks are expected to be 32-byte aligned in memory and be exactly 32 bytes long. The compiler breaks long basic blocks into chunks, using no-op instructions to pad the created chunks to their desired fixed length. The compiler adds “sanitizing” instructions which align targets of control-flow instructions (including returns) to guarantee that they branch to the beginning of a chunk, i.e., to a chunk-aligned address. (See § 4.2.2 for details.)

The verifier deems the loaded code safe by verifying each code chunk individually. For each chunk, the verifier verifies that if the chunk contains a control-flow instruction, it is the last one, and that it is sanitized as described above. This ensures verified code can only branch to verified code, keeping execution inside the sandbox. In addition, the verifier checks that memory operations are preceded, within the same chunk, by the instructions sanitizing their address operands and that the chunk does not contain privileged instructions.

3.3.3 Discussion: Privbox vs. NaCl

Our sandbox design is inspired by Native Client (NaCl) [13, 14]. NaCl enables safe execution of native code downloaded from a web site inside a web browser at near native speed. The browser verifies loaded programs and makes sure the loaded code does not write to, or jump, outside of a sandboxed region. The sandboxed code is loaded into a memory region that is gapped by unmapped memory regions on both ends. To ensure sandboxed code does not write or execute memory outside of the sandbox, NaCl relies on “offset-from-known-base” operations. A *base pointer register* (immutable by sandboxed code) points at this memory region. A memory access is allowed only with an offset from base pointer register, which results in accesses always being either: (1) inside the allowed memory region; or (2) in the unmapped memory areas.

Privbox’s instrumentation shares some similarities with

NaCl: (1) execution is limited to code running inside the sandbox; (2) memory stores have to be instrumented (because kernel memory is accessible); and (3) instructions have to be aligned in specific manner. In contrast to NaCl, however, Privbox (1) has no need for a base pointer register—we use absolute addresses, appropriately sanitized; (2) instruments memory loads as well, because kernel memory is readable; and (3) must avoid privileged instructions.

4 Implementation

This section describes our prototype implementation of the Privbox design in Linux. § 4.1 describes OS and library modifications and § 4.2 describes the sandboxing compiler. We do not implement the verifier part of the design (§ 3.3.2), as it is not required for evaluating performance under Privbox.

4.1 OS & library support

The following describe various parts of our implementation and their size in lines of code (LOC).

Semi-privileged execution (750 LOC) To implement semi-privileged execution, we apply the techniques of kernel-mode Linux (KML [7]) to Linux v5.8 on x86-64. KML is an existing kernel patch to support execution of an entire application in kernel mode. It is based on Linux v4.0 (circa 2015), and does not isolate its in-kernel processes from each other or the kernel from them.

Linux v5.8 on x86-64 uses “legacy stack switching,” where the stack is switched by the hardware only on a CPL change (i.e., an interrupt while user code is executing). However, Privbox’s semi-privileged execution has $CPL = 0$ but with a user stack. This means that an interrupt received during semi-privileged execution would cause the interrupt handler to run with the user’s stack, which is problematic because: (1) the user stack is accessible to user code, which might hijack execution by modifying the stack frame (from another thread); (2) writing to the stack might fault (e.g., if the stack pointer points to an unmapped page), but the page fault would not change the stack either, crashing the system; and (3) the user’s stack includes a red zone [26] that must not be written to. We therefore adjust our Linux version to use x86-64’s *interrupt stack table* (IST) for all interrupts and exceptions. With IST, each exception/interrupt/trap can be configured to switch to a specific stack.

Call gate (80 LOC) Privbox exposes a system call gate (§ 3.2), which is a kernel function that serves as the system call entry point for privboxed code. The gate follows Linux’s `syscall` conventions for passing the system call number and parameters. It is similar to the standard system call entry code but avoids performing unnecessary steps, such as modification of page tables and toggling of interrupts. In particular, Linux’s entry code (with PTI) assumes it is called from user-space and thus unconditionally switches the page

table from the user-space to the kernel page table. This is unnecessary for privboxed code, which already has kernel memory mapped in its custom page table.

Limitations For implementation simplicity, we inhibit receipt of signals during semi-privileged execution. This is not a design limitation, and there are several designs for supporting signals: (1) ensuring that the signal handler code points to verified and safe code; or (2) aborting privileged execution (i.e., having it return to the code that launched it with an `EINTR` indication). Importantly, privboxed code can still receive signals in our prototype using the `signalfd` [27] mechanism.

Library support (260 LOC) Applications usually invoke system calls through a C library function, which then invokes the `syscall` instruction. We thus create a modified standard C library, based on the `musl` C library [16], in which the library’s system call wrappers use `syscall` or Privbox’s system call gate based on the execution’s CPL. The entire library is compiled with Privbox’s instrumentation, so that privboxed code objects can be linked with it.

In this paper, we modify an application to use Privbox by changing its build environment to link privboxed code with the above C library. The reason is that current compilers and linkers do not support linking an entire application with both the system’s C library and our modified, instrumented C library, as both export the same symbols and the tools cannot resolve which library version the application code refers to. This problem can be solved by adding compiler annotations for demarcating privboxed code; we leave this to future work.

4.2 Code instrumentation

We implement Privbox’s sandboxing instrumentation by introducing a machine function pass and several other changes to the x86-64 backend of the LLVM toolchain [28], which consist of 1200 LOC. Our modified LLVM emits machine code in which unsafe instructions are replaced with equivalent but safe instruction sequences (§ 3.3).

Instrumented code is partitioned into fixed-size chunks (§ 3.3.2). Our implementation uses 32-byte chunks. The reason is that an x86-64 instruction can be up to 15 bytes long, so a 32-byte chunk can fit at least an instruction of the privboxed code plus the added instrumentation instructions that make it safe. (This is the worst case; most chunks contain more than one instruction.)

When instrumenting an instruction, it is placed in its own chunk, preceded with the instrumentation instructions, which is achieved by emitting an alignment directive in the code (`.align`). This ensures any instrumentation sequence starts at beginning of a new chunk.

4.2.1 Load/store instrumentation

Loads/stores are non-branching instructions that access memory. Their address operand is either static, verifiable at load

time, or dynamic, derived from values of registers. Dynamic values cannot be verified at load time, so instrumentation is required to ensure kernel memory is not accessed. Our instrumentation “sanitizes” operands by clearing their most significant bit (MSB), which ensures it does not point to kernel memory (§ 3.3).

On x86-64, memory operands are based on four elements: scale, index, base and displacement. Scale and displacement are scalars while index and base are registers. The effective address of a memory operand is calculated by: $Displacement + Base + Scale * Index$. Either the base or index registers can be omitted and are calculated as zero in such case. Scale can be 1, 2, 4 or 8. Displacement can be either 1, 2, 4 or 8 bytes long.

The memory operand of an instruction I is sanitized by the prefixing I with the following instruction sequence: (1) computing I ’s effective address with a load-effective-address instruction (`lea`); (2) clearing its MSB with a bit-test-and-reset instruction (`btr`); and (3) replacing I ’s original memory operand with one dereferencing the sanitized value. Listing 2 shows this sequence. The `btr` instruction has a side-effect of updating the x86-64 `EFLAGS` register. Our compiler code therefore checks if the `EFLAGS` register has meaningful state at the point of instrumentation, and if so, emits `SAVE_EFLAGS` and `RESTORE_EFLAGS` around the instrumentation sequence. These are abstract operations implemented by LLVM and translated to instructions such as `sahf/lahf` (save/load flags).

A shorter instrumentation sequence is used for memory operands that specify (1) only one of base or index registers; and (2) 1/2/4-byte displacement. In this case, the effective address is sanitized with a single `btr` instruction (and `EFLAGS` save/restore, if needed). Appendix A.1 provides the details.

Similarly to NaCl, we avoid instrumentation of stack loads/stores by maintaining and verifying invariants on manipulations of the stack pointer, which guarantee that stack accesses always target user memory. Appendix A.2 elaborates on handling of stack accesses. This approach greatly reduces the emitted instrumentation, as stack accesses are very common.

4.2.2 Control-flow instrumentation

Control-flow instructions are instructions that can modify the instruction pointer (beyond advancing it to next instruction). As with load/store instructions, we are concerned only with instructions whose operand is unknown at load time. Control-flow instructions can compromise safety of semi-privileged execution by branching to (1) arbitrary kernel code or (2) privboxed code at the middle of a chunk. The latter is dangerous because the verifier verifies code starting at chunk boundaries.

Similarly to load/store instrumentation, kernel addresses are avoided by clearing the MSB of branch targets. Chunk-unaligned addresses are avoided by clearing the low 5 bits of the target. While this allows branching to any chunk-aligned

```

.align CHUNK_SIZE
SAVE_EFLAGS
%Reg1 = lea disp(%Idx, scale, %Base)
%Reg2 = btr $63, %Reg1
RESTORE_EFLAGS
OP operand1, (%Reg2)

```

Listing 2: Instrumentation of load/store instruction with memory operand.

```

.align CHUNK_SIZE
pop %rcx
add $(CHUNK_SIZE - 1), %rcx
and $~(CHUNK_SIZE - 1), %rcx
btr $63, %rcx
jmp *%rcx

```

Listing 4: Instrumentation of return instruction.

address in user memory, only the privileged code section is mapped executable in the privboxed code’s page table (§ 3.3).

Return instrumentation A return is equivalent to popping an address from stack and jumping to it. To ensure a valid destination, we replace each return instruction with an equivalent but safe sequence that pops the address into a register, clears its MSB, aligns it to the next 32 bytes, and jumps to the obtained value. Listing 4 details this instrumentation sequence. Linux’s calling convention specifies that the RCX register is not preserved on calls, so we explicitly use it to store the return address. The `add` (addition) and `and` (logical AND) instructions are used to align-up the value in RCX to 32 bytes (start of next code chunk). The `btr` (bit-test-and-reset) clears the MSB.

This instrumentation ensures branching is possible only to code chunk aligned, non-kernel addresses. While in theory it is possible to hijack execution by overwriting the return address (e.g., by buffer overflow), the effects of such hijacking are very limited. Any address popped from the stack is guaranteed to be sanitized before use, so an attacker can only redirect execution to valid and verified code inside the Privbox code region or to non-executable memory (either user addresses or non-canonical). The former one does not pose a threat as privboxed code is verified as safe, and the latter causes a fault, effectively stopping the execution.

Call/jump instrumentation Call/jump instructions with an indirect destination (i.e., non embedded as an instruction-relative offset) can have their destination stored in one of two ways: (1) in a register operand and (2) in a memory operand.

Register operands are sanitized similarly to a return (sans stack pop), as shown in Listing 5. Since a return aligns addresses before branching, an alignment directive is required right after a call to push the next instruction to a chunk boundary. The EFLAGS register has to be preserved only in case of jumps, as the calling convention states that it is not preserved across calls.

```

.align CHUNK_SIZE
%Reg1 = lea *disp(%Idx, scale, %Base)
%Reg2 = btr $63, %Reg1
%Reg3 = mov *%Reg2
%Reg4 = btr $63, %Reg3
%Reg5 = and $~(CHUNK_SIZE - 1), %Reg4
call *%Reg5
.align CHUNK_SIZE

```

Listing 3: Instrumentation of memory-operand call.

```

.align CHUNK_SIZE
%Reg1 = btr $63, %Reg
%Reg2 = and $~(CHUNK_SIZE - 1), %Reg1
call *%Reg2
.align CHUNK_SIZE

```

Listing 5: Instrumentation of register-operand call.

A call/jump with a memory operand is equivalent to a memory-to-register load followed by a register operand call. Sanitization is this performed analogously: (1) the memory operand is sanitized with load/store instrumentation; (2) a `mov` instruction is used to load the address into a register; and (3) the loaded address is sanitized as a register operand call. Listing 3 shows the generated instruction sequence.

Jumps are similar to calls, except that they have to preserve the EFLAGS register and aligning the succeeding instruction is unnecessary, as jumps do not return. Appendix A.3 provides the details.

4.2.3 Code alignment

The instrumentation described in the previous sections deals with unsafe instructions, co-locating instrumentation sequences within same code chunks, and aligning return sites of calls. The compiler also makes sure that all branch destinations (functions, basic blocks) are aligned to the chunk boundary, because indirect branches target addressed with 5 lowest bits cleared. It additionally inserts no-op instructions before any instruction that would otherwise cross code chunk boundary, so that it moves to a chunk-aligned address. Combined, these rules partition the emitted code instructions into fixed-size chunks.

5 SPAP: Hardware support for reducing instrumentation overhead

Our analysis of Privbox’s performance (§ 7.3) shows that load/store instrumentation is responsible for a considerable part of instrumentation overhead. To address this problem, we propose *semi-privileged access prevention* (SPAP), a simple hardware architectural modification that enables omitting load/store instrumentation from privboxed code.

SPAP SPAP is a hardware feature that guarantees semi-privileged (privboxed) code cannot (1) read/write kernel memory nor (2) indirect branch to kernel memory. Of course, the CPU has no notion of “kernel memory” or “semi-privileged execution”—we define *kernel memory* as any virtual address mapped by a PTE with the ‘supervisor’ bit set, and *semi-privileged execution* as instructions executing in privileged mode (CPL = 0) but that are located in non-kernel memory (i.e., a clear ‘supervisor’ bit in the code page’s PTEs).

Assuming SPAP, it is possible to forgo all load/store instrumentation and limit control-flow instrumentation (§ 4.2) only to masking of jump targets to guarantee their chunk alignment. Listing 6 shows the simplified instrumentation call instruction enabled by SPAP (compared to Listing 3). The code chunk size

```

.align 16
%Reg1 = mov *disp(%Idx, scale, %Base)
%Reg2 = and $0xf, %Reg1
call *%Reg2
.align 16

```

Listing 6: Control-flow only instrumentation of memory-operand call.

is reduced to 16 bytes, which is enough to fit both the instrumented instructions and the required prefixes.

SPAP implementation We argue that SPAP can be implemented analogously to how current x86-64 processors implement SMAP/SMEP, which block *privileged mode execution* from accessing *user addresses*.³ Restricting privboxed code data accesses and branching can happen at the same pipeline stages that enforce SMAP and SMEP, respectively. Listing 7 describes how the hardware can restrict privboxed code’s data access: other than the CPL and PTE of the accessed page, which are already required by SMAP, SPAP only depends on the current instruction’s PTE, which is available in the instruction TLB. Similarly, Listing 8 shows how SPAP hardware restricts privboxed branching. The information required is same as what is needed for the SMEP mechanism, plus the information of whether the current instruction is an indirect branch.

Expected overhead We claim that SPAP’s additional access checks should have little to no effect on the latency of memory instructions. We base this claim on the overhead observed from enabling SMAP/SMEP, shown below, and the similarity of SPAP to them.

We evaluate SMAP overhead (on the platform described in § 7). We measure average load latency when accessing differently sized working sets from kernel space, with and without SMAP. Our test traverses each cache line in the working set buffer in random order (to prevent prefetching), with each load depending on the result of the previous one (to prevent the CPU’s out-of-order execution from overlapping load execution). Preemption is disabled during the test, to ensure it has exclusive use of the CPU. We measure average cycles per load (i.e., total number of cycles divided by number of loads performed). Each test is run 31 times and we report the average of the last 30 runs.

Figure 3 shows results for working set sizes targeting the capacity of the CPU’s TLB and L1/L2/L3 caches. We find that SMAP does not impact performance in a significant way, as (1) some tests still execute faster with SMAP enabled; and (2) the variance is greater than the difference between the configurations.

```

if (
  CPL < 3 and
  AccessedPage.S_bit is Set and
  CurrInstPage.S_bit is Cleared
):
  trap()

```

Listing 7: Hardware enforcement that semi-privileged execution loads/stores do not access kernel memory.

```

if (
  CurrInst is Indirect Branch and
  CPL < 3 and
  FetchPage.S_bit is Set and
  CurrInstPage.S_bit is Cleared
):
  trap()

```

Listing 8: Hardware enforcement that semi-privileged execution does not indirect branch to kernel code.

³The idea is to prevent exploits of kernel memory safety bugs that attempt to, e.g., jump to user-space code [15].

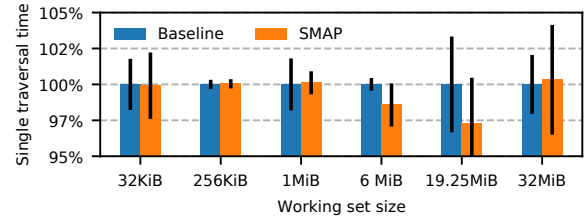


Figure 3: SMAP overhead on load instructions. Results are normalized to execution with SMAP disabled.

6 Security

We analyze Privbox’s security against architectural and microarchitectural (transient execution) attacks. Recall that in privboxed code, every chunk (a 32-byte range at a 32-byte aligned address) contains a correctly instrumented, non-privileged instruction sequence of exactly 32 bytes, perhaps ending with no-ops (see § 3.3.2). This is verified by the kernel.

Semi-privileged execution has the following invariants:

- Inv1** The target of *any* load/store instruction is not a kernel address. (Enforced by the instrumentation; § 4.2.1.)
- Inv2** The target of *any* control-flow instruction (including returns) is a 32-byte aligned non-kernel address. (Enforced by the instrumentation; § 4.2.2.)
- Inv3** The privboxed code section is read-only and user-space addresses outside of it are non-executable during semi-privileged execution. (Enforced by the virtual memory permissions; §§ 3.2–3.3.)

For normal (non-transient) instruction execution, Inv2 and Inv3 imply that semi-privileged execution can run only (instrumented) code located in the privboxed region. By Inv1, such code cannot access kernel memory, and the verifier guarantees it does not contain privileged instructions. Therefore, if regular unprivileged execution cannot perform some operation or memory access, neither can semi-privileged execution.

It remains to analyze transient execution attacks. Generally, such an attack uses architecturally-incorrect flows (whose instructions execute but do not subsequently commit) to leak memory contents via a microarchitectural side-channel [29]. Privbox’s goal is thus to protect kernel memory.

The core observation is that transient execution of a *complete instrumented* chunk is safe, because it still sanitizes the operands of any memory operation in the chunk. We therefore only need to consider if transient execution can branch mid-chunk or outside of the privboxed region, either of which can happen due to indirect branch or return target misprediction. Crucially, we consider *any* “supervisor mode” transient execution—both semi-privileged and standard kernel execution—to cover attacks of privboxed code on the kernel. To this end, we analyze how the branch predictor can be “trained,” i.e., which targets it observes and may mispredict execution to later:

We assume any training by user-space execution cannot affect “supervisor mode” execution, due to existing mitigations

such as Intel’s enhanced indirect branch restricted speculation (eIBRS) [30] or Arm’s CSV2 [31]. If this assumption does not hold, then the kernel is vulnerable regardless of Privbox.

We assume kernel execution can only train valid kernel branch/return targets (and therefore the kernel cannot transiently branch to privboxed code), because if an attacker can cause the kernel to train arbitrary addresses, they can attack the kernel regardless of Privbox. This still means that semi-privileged execution may (transiently) branch to a valid kernel function, possibly creating a speculative type confusion vulnerability [32]. Privbox’s instrumentation can mitigate this problem (with run-time overhead) using *retpolines* [33] for indirect branches.

Finally, by *Inv2*, semi-privileged execution can only train non-kernel, chunk-aligned addresses. It might thus train user-mode addresses outside of the privboxed region. We assume that instructions from non-executable memory are not executed in transient execution,⁴ so by *Inv3*, semi-privileged execution cannot be exploited by such training. However, training by semi-privileged can cause subsequent kernel execution to (transiently) branch to user-space instructions and execute them, as Privbox disabled SMEP. On current hardware, training by semi-privileged execution can be prevented from affecting the kernel’s execution using a mechanism such as Intel’s indirect branch predictor barrier (IBPB) [36] in the Privbox system call gate, but this will slow down system calls in privboxed code. (Our Privbox prototype does not implement this mitigation.) Future hardware could support SPAP-like extensions to eIBRS to make predictions of branches executed from supervisor pages uncontrollable by branches executed from user-level pages.

We assume SPAP can be implemented so that its restrictions apply to both normal and transient execution, as SMAP/SMEP have this guarantee [34, 35].

Limitation: Disabled SMEP Privbox’s security drawbacks stem from disabling SMEP for semi-privileged execution without re-enabling it for kernel execution, as Privbox does for SMAP. As a result, semi-privileged execution can (1) exploit pre-existing kernel vulnerabilities that were mitigated by SMEP and (2) mount transient execution attacks against the kernel, as explained above. The SMEP limitation can be addressed by hardware reducing the cost of toggling SMEP. This should be possible, given that hardware has optimized SMAP toggling, an action the kernel frequently performs.

7 Evaluation

We evaluate the impact of Privbox on system call latency (§ 7.1), on system call-intensive I/O threads (§ 7.2), and the impact of privboxing complete real-world applications (§ 7.3).

⁴This holds on x86-64, where documentation states that SMEP and virtual memory execute restrictions apply to transient execution [34, 35].

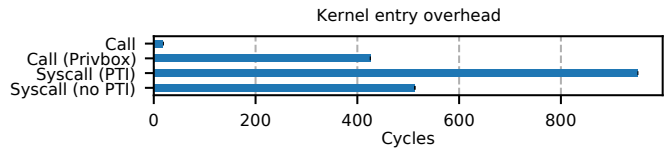


Figure 4: Cycles taken to execute a roundtrip to kernel using different entry methods. Regular call added for reference.

Platform We use a Dell PowerEdge R740 server with a 28-core Intel Xeon Gold 6132 (Skylake) CPU and 192 GiB of DRAM. Hyper-Threading is disabled. Due to current cloud computing trends, virtualized platforms represent the environments where evaluated workloads usually run. We therefore use a Linux v5.8 guest in a KVM virtual machine hosted on a Ubuntu 18.04 host. Reported measurements are averages of 10 executions after a single warmup run; error bars indicate standard deviation.

7.1 System call latency

We measure the end-to-end latency of invoking a non-existing system call, which covers user-to-kernel transition, entry code execution, and kernel-to-user return (with a “bad call” error). Our benchmark invokes the system call 100 M times and reports average invocation latency, measured with the CPU’s cycle counter.

We compare the latency of a regular system call invocation with and without PTI to the latency of invoking the system call from within privboxed code. Figure 4 shows that a system call invocation alone takes about 950 and 510 cycles with and without PTI, respectively. Invocation from privboxed code takes on average 425 cycles, 2.2× and 1.2× faster, respectively, than the baseline with and without PTI.

The reason that a privboxed system call invocation is slower than a regular function call is that while Privbox eliminates hardware user/kernel transition costs, it must still manage software-related user/kernel transition steps. For instance, Privbox’s system call gate (§ 3.2) switches the stack and saves/restores register state.

7.2 I/O-thread workloads

Here, we characterize the impact of Privbox on an I/O thread-based application architecture (see § 3.1 and Figure 1). We benchmark a generic server program that receives requests, processes them, and returns response. The server is composed of I/O and compute threads, which are responsible, respectively, for socket operations and the “business logic” of computing responses to incoming requests.

Our benchmark has two tunable parameters: (1) *Compute time*, the time compute thread spends on each request, which allows controlling how compute-heavy the workload is; and (2) *I/O size*, the number of bytes each for each socket I/O operation. We use fixed-sized messages, so the I/O size determines the number of system calls per message, i.e., how system call-intensive the workload is.

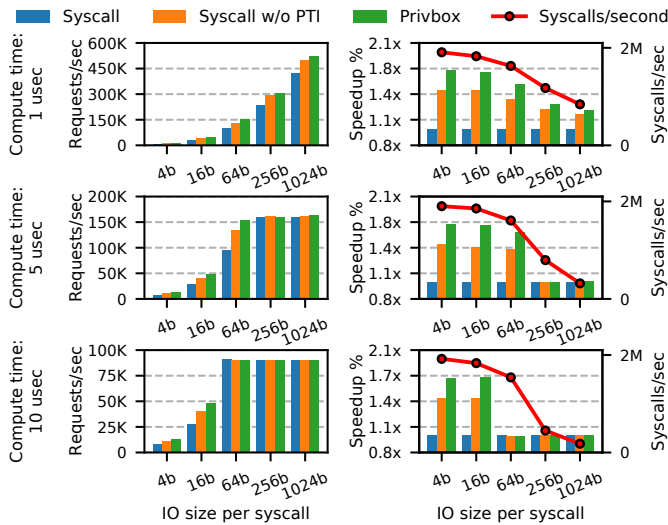


Figure 5: Privbox impact on an I/O-thread based server. Left: absolute throughput (requests/second), right: throughput normalized to default Linux (with PTI). Rows describe different compute times.

We compare between Privbox and standard execution with PTI (default Linux configuration) and without it. When using Privbox, only the I/O thread code is privboxed (and therefore compiled with instrumentation). Figure 5 shows the server’s throughput (requests/second) as we vary the compute time (across rows) and I/O size (X axis in each row). For large I/O sizes (above 256 bytes) and compute time (above 5 μ s), system call invocation frequency decreases and so all kernel entry methods yield similar throughput, as most CPU time is spent on compute or inside system calls (waiting for I/O). However, for fast compute and/or high rate of system calls (small I/O size), Privbox results in up to $1.72\times$ speedup compared to regular execution with system calls.

7.3 Real-world workloads

This section analyzes the impact of Privbox on several popular real-world applications: Redis [18], memcached [19], and SQLite [20]. We modify each application to use Privbox, which requires changing/adding about 20–30 lines of code to make the application’s main loop execute privboxed. All binaries are compiled with `-O2` optimizations and linked with our instrumented `musl-1.2.0` C library. Importantly, we compile the entire application with Privbox’s instrumentation, not only the part that runs privboxed. The reason is that our Privbox prototype does not support compiling an application with both instrumented and uninstrumented versions of the C library (see § 4.1). The upshot is that our results here are *lower bounds* of Privbox’s benefit, as we instrument code that a full Privbox implementation would not.

To analyze instrumentation overhead, we measure each application with three instrumentation levels: (1) No instrumentation (*noinstr*), which shows the benefit from fast system call invocation; (2) full instrumentation (*fullinstr*, § 4.2), which shows Privbox’s benefit (faster system calls, but with instru-

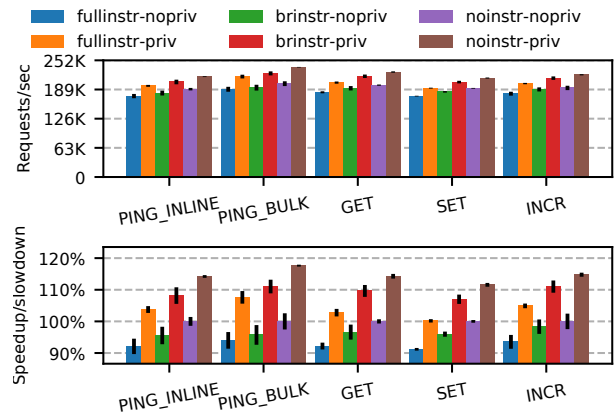


Figure 6: Redis server under load of redis-benchmark running 2 threads, 50 connections, 1 M requests. Top: throughput (requests/seconds). Bottom: throughput relative to ‘noinstr-nopriv’.

mentation overhead) on current hardware; and (3) control-flow only instrumentation (*brinstr*, § 5), which omits load/store instrumentation, thereby emulating Privbox’s benefit on hardware with SPAP support.

To analyze the benefit from Privbox’s fast system calls, we measure each instrumentation level with and without executing the privboxed code sections in privileged mode (tagged *priv* and *nopriv*, respectively). The speedup of *priv* over *nopriv* quantifies how Privbox’s fast system calls offset instrumentation overhead.

Redis Redis [18] is a popular key-value store, often used as cache, document store, or for publish/subscribe messaging. We use Redis’ recommended default setup of a single instance running a single thread, without persistency. We modify Redis’ main loop to execute privboxed. The privboxed loop returns to user-space once per 10 K iterations to service signals (due to limitations of our prototype, see § 4.1).

We evaluate Redis using two benchmarks: (1) *redis-benchmark*, with which we simulate running various Redis commands by 50 concurrent clients that send 1 M requests, and (2) *memtier_benchmark* [37], a stress tester for NoSQL databases, which we run with a 10:1 read/write ratio of 32-byte objects.

Figure 6 shows redis-benchmark throughput of various Redis commands. Reducing system call overhead offers significant benefit: ‘priv’ executions have on average 13% higher throughput than their ‘nopriv’ variants. While Privbox’s instrumentation overhead offsets some of this benefit, overall, a Privboxed Redis (‘fullinstr-priv’) obtains up to 7.6% higher throughput than its baseline (‘noinstr-nopriv’). Had the CPU supported SPAP (enabling less instrumentation: ‘brinstr-priv’), the throughput would improve to up 10% higher than the baseline. Results from *memtier_benchmark* (Figure 7) show similar trends, with ‘fullinstr-priv’ and ‘brinstr-priv’ obtaining 6% and 10% higher throughput than the ‘noinstr-nopriv’ baseline.

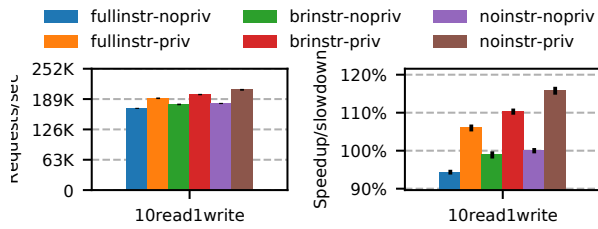


Figure 7: Redis server under load of memtier_benchmark, running 4 threads, 50 clients/thread, 10 K requests/client, no pipelining, 32 byte objects, 10:1 read/write ratio. Left: throughput (requests/seconds). Right: throughput relative to ‘noinstr-nopriv’.

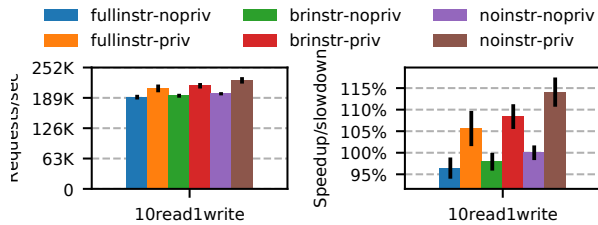


Figure 8: Memcached under load of memtier_benchmark, running 4 threads, 50 clients/thread, 10 K requests/client, no pipelining, 32 byte objects, 10:1 read/write ratio. Left: throughput (requests/seconds). Right: throughput relative to ‘noinstr-nopriv’.

Memcached memcached [19] is a distributed object caching system offering a key-value store interface. Similarly to Redis, we modify the main loop to execute privboxed, and break out to user-space once per 10 K iterations.

We evaluate memcached by measuring the throughput obtained by memtier_benchmark, again with a 10:1 read/write ratio and keys/values of 16/32 bytes, respectively. Figure 8 shows the results, which mirror those of Redis. Specifically, ‘priv’ executions have on average 19% higher throughput than their ‘nopriv’ variants, which is sufficiently high for Privbox to outperform the baseline: ‘fullinstr-priv’ and ‘brinstr-priv’ obtain 4.5% and 6.9% higher throughput than the baseline.

SQLite SQLite [20] is a relational database engine. We evaluate it using sqlite-bench [38], a tool that measures throughput of various access patterns: writing/reading of sequential/random values in asynchronous/synchronous/batched modes. We use a RAM filesystem (tmpfs) to store the database files.

Figure 9 shows throughput obtained for each access sequence. Many sequences stand to benefit from Privbox’s fast system calls (evidenced by an average 8% speed up of ‘priv’ over ‘nopriv’ variants), but these benefits are negated by instrumentation overhead. However, some patterns (“readrandom” and “fillseqsync”) do not benefit from fast system calls.

Figure 10 explains the above results. It shows the ratio between number of system calls invoked to time spent in user code (i.e., CPU time minus system call execution time). We find a strong correlation between speedup from fast system calls and the system call/user time ratio. For example, “readrandom” suffers greatly from instrumentation because SQLite performs read queries using loads/stores (which are instrumented) without invoking system calls. The “fillseq-

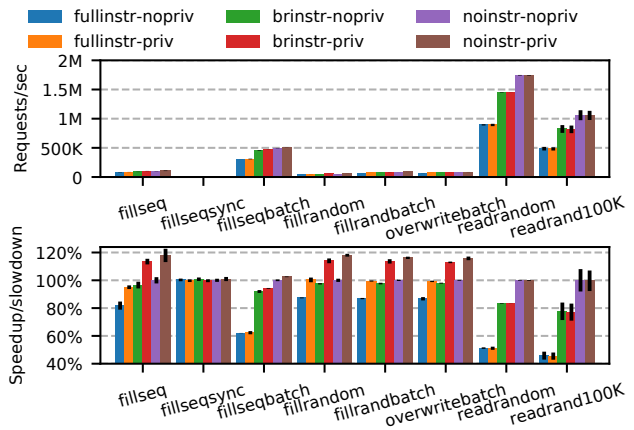


Figure 9: SQLite throughput. Top: throughput (operations/second). Bottom: throughput relative to ‘noinstr-nopriv’.

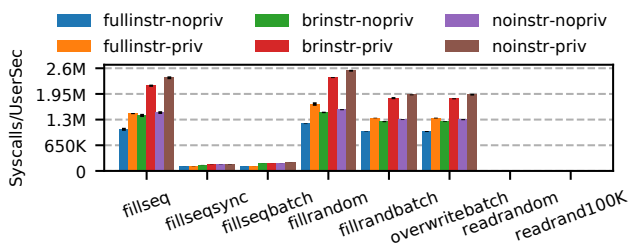


Figure 10: SQLite ratio of system calls to time spent in user-space code (log scale). Correlation can be seen with the bottom half of Figure 9: tests with a high system call to user time ratio show better speedup from ‘priv’ execution.

batch” sequence, which behaves the same with and without fast system calls (‘priv’), batches I/O operations and thus performs fewer system calls. Finally, the “fillseqsync” sequence, which behaves the same for all instrumentation levels and execution modes, uses slow synchronous writes and so spends most of its time waiting, as opposed to running user code or entering/exiting the kernel.

8 Conclusion & future work

We propose Privbox, a design for speeding up system calls by sandboxed semi-privileged execution, without changing the underlying system call programming model. We believe Privbox can also be useful to improve isolation and fault tolerance within the kernel, e.g., by privboxing modules and device drivers to limit the memory and kernel APIs they access.

Our Privbox prototype uses simple compile-time instrumentation which incurs non-negligible overhead, offsetting some of the benefit from Privbox’s fast system call invocation. There are several directions for reducing instrumentation overhead, which we leave to future work: Hardware features such as Intel’s control-flow enforcement technology (CET [39]) can be useful for reducing control-flow instrumentation. Finally, a more sophisticated verifier can avoid redundant instrumentation (e.g., sanitizing a previously-sanitized register).

Acknowledgments

We thank the paper’s anonymous reviewers and shepherd for their feedback and suggestions, which helped strengthen the paper. We also thank the artifact reviewers for their careful work, which helped improve the artifact.

References

- [1] Livio Soares and Michael Stumm. “FlexSC: Flexible System Call Scheduling with Exception-Less System Calls.” In: *OSDI*. 2010.
- [2] Lars Müller. “KPTI a mitigation method against meltdown”. In: *Advanced Microkernel Operating Systems* (2018), p. 41.
- [3] *Fighting Spectre with cache flushes*. <https://lwn.net/Articles/768418/>.
- [4] Moritz Lipp et al. “Meltdown: Reading kernel memory from user space”. In: *USENIX Security*. 2018.
- [5] Paul Kocher et al. “Spectre attacks: Exploiting speculative execution”. In: *IEEE SP*. 2019.
- [6] Mohan Rajagopalan et al. “Cassyopia: Compiler Assisted System Optimization”. In: *HotOS*. 2003.
- [7] Toshiyuki Maeda. “Kernel Mode Linux”. In: *Linux J*. 2003.109 ().
- [8] *Efficient io with io_uring*. https://kernel.dk/io_uring.pdf.
- [9] *DPDK: Data Plane Development Kit*. <https://www.dpdk.org/>.
- [10] Jonathan Behrens et al. “Efficiently mitigating transient execution attacks using the unmapped speculation contract”. In: *OSDI*. 2020.
- [11] Paul Barham et al. “Xen and the Art of Virtualization”. In: *SIGOPS Oper. Syst. Rev.* 37.5 ().
- [12] *VMWare: Paravirtualization API Version 2.5*. https://www.vmware.com/pdf/vmi_specs.pdf.
- [13] Bennet Yee et al. “Native client: A sandbox for portable, untrusted x86 native code”. In: *IEEE SP*. 2009.
- [14] David Sehr et al. “Adapting Software Fault Isolation to Contemporary CPU Architectures”. In: *USENIX Security*. 2010.
- [15] *Supervisor mode access prevention*. <https://lwn.net/Articles/517475/>.
- [16] *musl C library*. <https://www.musl-libc.org/>.
- [17] Edited Jim Coplien and Douglas C Schmidt. “Reactor-an object behavioral pattern for demultiplexing and dispatching handles for synchronous events”. In: (1995).
- [18] *Redis*. <https://redis.io/>.
- [19] *memcached*. <https://memcached.org/>.
- [20] *SQLite*. <https://www.sqlite.org/index.html>.
- [21] *SPDK: Storage Performance Development Kit*. <https://spdk.io/>.
- [22] Yuhong Zhong et al. “BPF for Storage: An Exokernel-Inspired Approach”. In: *HotOS*. 2021.
- [23] *eBPF*. <https://ebpf.io/>.
- [24] A Starovoitov, J Schulist, and D Borkmann. “Linux Socket Filtering aka Berkeley Packet Filter (BPF)”. In: *Documentation/networking/filter.txt* (2016).
- [25] Matt Welsh, David Culler, and Eric Brewer. “SEDA: An Architecture for Well-Conditioned, Scalable Internet Services”. In: *SOSP*. 2001.
- [26] Michael Matz et al. “System V Application Binary Interface”. In: *AMD64 Architecture Processor Supplement, Draft v0 99* (2013), p. 57.
- [27] *Signalfd manual page*. <https://man7.org/linux/man-pages/man2/signalfd.2.html>.
- [28] *LLVM project*. <https://llvm.org/>.
- [29] Claudio Canella et al. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: *USENIX Security*. 2019.
- [30] Intel. *Indirect Branch Restricted Speculation*. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html>.
- [31] Arm. *Vulnerability of Speculative Processors*. <https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability>.
- [32] Ofek Kirzner and Adam Morrison. “An Analysis of Speculative Type Confusion Vulnerabilities in the Wild”. In: *USENIX Security*. 2021.
- [33] *Retpoline: a software construct for preventing branch-target-injection*. <https://support.google.com/faqs/answer/7625886>.
- [34] AMD. *Software Techniques For Managing Speculation On AMD Processors*. <https://www.amd.com/system/files/documents/software-techniques-for-managing-speculation.pdf>.
- [35] Intel. *Speculative Execution Side Channel Mitigations*. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html>.

- [36] *Indirect Branch Predictor Barrier*. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html>.
- [37] *memtier-benchmark*. https://github.com/RedisLabs/memtier_benchmark.
- [38] *sqlite-bench*. <https://github.com/ukontainer/sqlite-bench>.
- [39] *Intel CET*. <https://newsroom.intel.com/editorials/intel-cet-answers-call-protect-common-malware-threats/>.
- [40] *privbox/devenv: ATC'22*. <https://doi.org/10.5281/zenodo.6618853>. DOI: 10.5281/zenodo.6618853.
- [41] *privbox/linux: ATC'22*. <https://doi.org/10.5281/zenodo.6618867>. DOI: 10.5281/zenodo.6618867.
- [42] *privbox/musl: ATC'22*. <https://doi.org/10.5281/zenodo.6618859>. DOI: 10.5281/zenodo.6618859.
- [43] *privbox/llvm-project: ATC'22*. <https://doi.org/10.5281/zenodo.6618847>. DOI: 10.5281/zenodo.6618847.
- [44] *privbox/redis: ATC'22*. <https://doi.org/10.5281/zenodo.6618855>. DOI: 10.5281/zenodo.6618855.
- [45] *privbox/memcached: ATC'22*. <https://doi.org/10.5281/zenodo.6618874>. DOI: 10.5281/zenodo.6618874.
- [46] *privbox/sqlite-bench: ATC'22*. <https://doi.org/10.5281/zenodo.6618869>. DOI: 10.5281/zenodo.6618869.
- [47] *privbox/piotbench: ATC'22*. <https://doi.org/10.5281/zenodo.6618857>. DOI: 10.5281/zenodo.6618857.
- [48] *privbox/libevent: ATC'22*. <https://doi.org/10.5281/zenodo.6618872>. DOI: 10.5281/zenodo.6618872.

A Instrumentation details

A.1 Special case load/store instrumentation

Listings 9 and 10 detail instrumentation for non stack-relative operations which have only one of the index/base registers specified, and displacement is either 1, 2 or 4 bytes long. In these scenarios, we can ensure any provided address value will become either a user or a non-canonical address by clearing bit 60 of the specified (base or index) register. This is sufficient because neither multiplication (in case of index register) nor addition of a 4-byte long displacement value will result in a canonical kernel address with 1s in all 16 most significant bits.

```
.align CHUNK_SIZE
SAVE_EFLAGS
%Reg1 = btr $60, %Base
RESTORE_EFLAGS
OP operand1, disp(, %Reg1)
```

Listing 9: Instrumentation of operand containing base register

```
.align CHUNK_SIZE
SAVE_EFLAGS
%Reg1 = btr $63, %Reg
%Reg2 = and $~(CHUNK_SIZE - 1),
        %Reg1
RESTORE_EFLAGS
jmp *%Reg2
```

Listing 11: Instrumentation of register operand jump

```
.align CHUNK_SIZE
SAVE_EFLAGS
%Reg1 = btr $60, %Idx
RESTORE_EFLAGS
OP operand1, disp(%Reg1, scale,)
```

Listing 10: Instrumentation of operand containing index register

```
.align CHUNK_SIZE
SAVE_EFLAGS
%Reg1 = lea *disp(%Idx, scale, %Base)
%Reg2 = btr $63, %Reg1
%Reg3 = mov *%Reg2
%Reg4 = btr $63, %Reg3
%Reg5 = and $~(CHUNK_SIZE - 1),
        %Reg4
RESTORE_EFLAGS
jmp *%Reg5
```

Listing 12: Instrumentation of memory-operand jump

A.2 Stack accesses

Stack-based operations can be considered safe as long as we ensure that at any point in time, the stack pointer points to valid user memory. The safety of stack-relative operations is ensured by maintaining the following invariant:

- When entering semi-privileged execution, the stack pointer must be set to a known valid value.
- When the stack pointer is set to a specific value, i.e. copied from another register, the copied value must be sanitized in a similar manner to an operand of a load/store instruction (i.e., clear its MSB).
- Each operation modifying/incrementing/decrementing the stack pointer must change the value by no more than a page, and must access the memory pointed by the new stack pointer value unconditionally afterwards (e.g., in same basic block). This permits operations like `push` and `pop`, as well as operations such as `add` and `sub`, as long as the memory is accessed through the stack pointer shortly after.

Incrementing/decrementing stack pointer without dereferencing can expose the code to an attack where the same sequence of instructions is used to modify stack pointer in small increments to an arbitrary value, until it points to kernel memory. Enforcing a stack access after the stack pointer changes makes sure that the stack pointer does not travel over inaccessible memory, such as the gap between kernel and user memory and the zero page in user memory, thereby preventing the stack pointer from overflowing/underflowing into kernel memory.

The above restrictions ensure that stack pointer always points to user memory, so loads/stores relative to the stack pointer register can be considered safe, as long as verifier successfully verifies that the above invariants hold.

A.3 Jump instructions

Listings 11 and 12 describe instrumentation of register and memory operand jumps, as mentioned in § 4.2.2.

B Artifact Description

Abstract

Our artifacts include all of the Privbox prototype code, as well as the scripts and benchmarks used to produce the results presented in this paper.

Scope

The artifacts can be used to:

- Set up a development and runtime environment for our prototype (§ 4).
- Run the experiments described in § 7, specifically, to reproduce results we present in Figures 3–9.

Refer to the artifact’s README (<https://github.com/privbox/devenv/blob/privbox/README.md>) for complete instructions.

Contents

- **devenv** [40] - a repository containing a README and scripts to set up a development and evaluation environment for the Privbox prototype.
- The Privbox prototype, which consists of:
 - **Linux** [41] and **musl C library** [42] - Operating system and C library with Privbox support.
 - **LLVM** [43] - LLVM toolchain capable of creating binaries instrumented for Privbox.
- **Benchmarks** [44, 45, 46, 47] - programs we used to evaluate Privbox.

Hosting

Our artifacts are available on Github (<https://github.com/privbox/>), as well as archived on Zenodo [40, 48, 41, 43, 45, 42, 47, 44, 46].

Requirements

Evaluation of our artifact requires an Intel x86-64 machine running Linux (we have used Ubuntu 18.04). Additionally, we rely on Docker and QEMU/KVM.

- **CPU type:** Our evaluation uses an Intel Skylake CPU. While any modern Intel-architecture CPU is suitable, evaluation results might differ due to microarchitectural changes.

- **Virtualization:** Our prototype runs as a KVM-based virtual machine. In our evaluation, we use a bare-metal server as a platform. It is possible to use a virtual machine, as long as it supports nested virtualization. However, nested virtualization incurs additional overhead that might affect evaluation results.



BBQ: A Block-based Bounded Queue for Exchanging Data and Profiling

Jiawei Wang^{1,2,3}, Diogo Behrens^{1,2}, Ming Fu^{1,2,*}, Lilith Oberhauser^{1,2}, Jonas Oberhauser^{1,2},
Jitang Lei^{1,2}, Geng Chen², Hermann Härtig³, and Haibo Chen^{2,4}

¹*Huawei Dresden Research Center* ²*Huawei OS Kernel Lab*

³*Technische Universität Dresden* ⁴*Shanghai Jiao Tong University*

Abstract

Concurrent bounded queues have been widely used for exchanging data and profiling in operating systems, databases, and multithreaded applications. The performance of state-of-the-art queues is limited by the interference between multiple enqueues (enq-enq), multiple dequeues (deq-deq), or enqueues and dequeues (enq-deq), negatively affecting their latency and scalability. Although some existing designs employ optimizations to reduce deq-deq and enq-enq interference, they often neglect the enq-deq case. In fact, such partial optimizations may inadvertently increase interference elsewhere and result in performance degradation.

We present Block-based Bounded Queue (BBQ), a novel ringbuffer design that splits the entire buffer into multiple blocks. This eliminates enq-deq interference on concurrency control variables when producers and consumers operate on different blocks. Furthermore, the block-based design is amenable to existing optimizations, *e.g.*, using the more scalable *fetch-and-add* instruction. Our evaluation shows that BBQ outperforms several industrial ringbuffers. For example, in single-producer/single-consumer micro-benchmarks, BBQ yields 11.3x to 42.4x higher throughput than the ringbuffers from Linux kernel, DPDK, Boost, and Folly libraries. In real-world scenarios, BBQ achieves up to 1.5x, 50.5x, and 11.1x performance improvements in benchmarks of DPDK, Linux `io_uring`, and Disruptor, respectively. We verified and optimized BBQ on weak memory models with a model-checking-based framework.

1 Introduction

Concurrent bounded queues are pervasive in operating systems, databases, and multithreaded applications. They transport data, distribute work, and are used to profile and decouple components. Their performance is crucial for achieving highly scalable and low-latency operation of numerous systems.

The main factor determining performance of a queue is the interference between concurrent operations, *i.e.*, between

enqueues, between dequeues, or between enqueues and dequeues. We refer to these as *enq-enq*, *deq-deq*, and *enq-deq* interference, respectively. Interferences manifest in the form of 1) cache-line bouncing when control variables are frequently updated by one thread and read by another, *e.g.*, to check if the queue has data, and 2) serialization of contended updates to control variables, *e.g.*, when multiple threads try to create or read the same entry. Existing queue designs often employ optimizations to reduce enq-enq and deq-deq interference, *e.g.*, updating control variables with “always-successful” atomic instructions such as *fetch-and-add* (FAA) [14, 40, 41, 47] because, in principle, they can be serialized in hardware and thus perform better under high contention than software solutions with *compare-and-swap* (CAS) [38, 43]. However, existing designs tend to neglect the enq-deq interference even though it substantially impacts performance, in particular in the common single producer or single consumer scenarios, *e.g.*, ringbuffers for asynchronous I/O in Linux `io_uring` [13].

In fact, some queue optimizations from the literature [38, 40] inadvertently increase the enq-deq interference or introduce undesirable side-effects that degrade performance in uncontended cases. For example, dequeue operations using FAA must either block in a pessimistic way [14], or risk overtaking slow concurrent enqueue operations; to avoid data corruption, such enqueue operations must be invalidated and repeated later [40]. Besides harming performance, such strategies cannot be applied for online profiling where enqueues writing a log should not be delayed by dequeues that read the log. Similarly, some techniques that avoid concurrent enqueue operations from waiting for each other also require dequeue operations to invalidate parts of the queue [38]. Strategies to improve performance of these techniques by reducing the number of invalidations, *e.g.*, busy-looping before invalidating [38, 40], drastically increase the latency of dequeue calls on empty queues, making them unsuitable for certain workloads, *e.g.*, multiplexing across multiple message queues.

We present Block-based Bounded Queue (BBQ), a novel ringbuffer design that dramatically reduces the enq-deq interference by splitting the entire buffer into multiple blocks and splitting the control variables into the block-level and queue-level variables. In the common case, enqueue and de-

*Ming Fu (ming.fu@huawei.com) is the corresponding author.

queue only access block-level control variables of their current blocks. When enqueue and dequeue work on different blocks, the disjoint control variables avoid any interference between these operations. That is particularly important in reducing the cache-line bouncing of head and tail pointers when determining whether the queue is full or empty. Furthermore, we use hardware-serialized `FAA` operations to update block-level control variables for allocating entries inside blocks, while queue-level control variables on the other hand are updated with slower, software-serialized `CAS` operations; since this is only necessary in the rare event that operations need to move to the next block, the performance impact of these operations is negligible. Our block-based approach allows us to perform these optimizations without incurring undesirable side-effects. Finally, to ensure that BBQ correctly works on weak memory models (WMMs) — including those from Arm [2] and RISC-V [28] architectures — we have verified and optimized the barriers and fences of BBQ with the `VSYNC` framework [42].

In contrast to previous work, our block-based approach is applicable to a large spectrum of scenarios. BBQ supports single or multiple producers/consumers, fixed- or variable-sized entries, and retry-new and drop-old modes. Retry-new is the typical producer-consumer mode for message passing and work distribution scenarios; drop-old is a lossy/overwrite mode for profiling/tracing [5, 24] and debugging [44] scenarios, in which producers may overwrite unconsumed data if the buffer is full.

In our experimental evaluation, BBQ outperforms several industrial queues and ringbuffers. In single-producer/single-consumer micro-benchmarks, BBQ yields 11.3x to 42.4x higher throughput than Linux circular buffer [22], DPDK ring buffer [9], Boost lock-free queue [4], and Meta’s Folly queue [14]. In real-world scenarios, BBQ achieves up to 1.5x, 50.5x, and 11.1x performance improvements in benchmarks of DPDK, Linux `io_uring` [13], and LMAX Disruptor [23], respectively. In our profiling benchmarks, BBQ enabled with the lossy/overwrite mode achieves up to 4.7x higher throughput than Google’s Guava `EvictingQueue` [15] and Apache Commons `CircularFifoQueue` [1], and can sustain up to 143.2x lower enqueue latency than the other two queues.

The remainder of this work is organized as follows. In Sec. 2, we gradually introduce the challenges of reducing the interference between enqueue and dequeue operations, discussing how existing queues tackle these challenges, and the limitations of their solutions. In Sec. 3, we present our block-based approach and the high-level design of BBQ. In Sec. 4, we describe BBQ implementation including the support for retry-new and drop-old modes and variable-sized entries. In Sec. 5, we report our results in verifying BBQ on WMMs and relaxing its memory barriers. In Sec. 6, we experimentally compare the performance of BBQ and several industry-grade concurrent queues. In Sec. 7, we conclude our work.

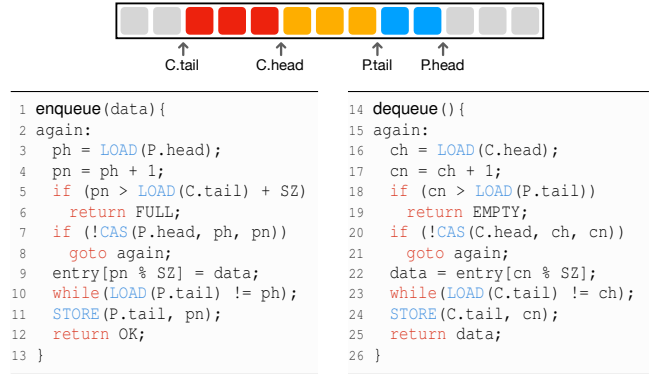


Figure 1: A simple MPMC bounded queue. `CAS`, `LOAD`, and `STORE` are atomic operations with sequentially consistent semantics on WMMs. `C.head` and `C.tail` refer to consumers; `P.head` and `P.tail` refer to producers.

2 Background and Related Work

We now introduce scalability challenges of bounded queue designs and discuss related work and their limitations. Figure 1 illustrates the discussion in this section depicting a simple lockless bounded queue with multiple-producer multiple-consumer (MPMC) support, which is the algorithm behind the widely-used DPDK ringbuffer [9].

Producers first check whether the queue is full (Line 5) and then try to *allocate* the next entry via `CAS` (Line 7). Upon success, the producer copies the data into the entry and *commits* it (Line 11). Similarly, consumers first try to *reserve* an entry (Line 20). Upon success, the consumer copies the data back and confirms that the data has been *consumed* (Line 24).

(P1) Consumer contention on C.head. A straightforward form of deq-deq interference is caused by multiple consumers concurrently calling dequeue and contending on updates to `C.head`. Several works (on bounded and unbounded queues) tackle this contention using `FAA` to update the head [38, 40, 43] because `FAA` is more scalable than `CAS` on common architectures. However, since `FAA` cannot conditionally update the memory location, it may break, for example, the invariant of `C.head` never exceeding `P.tail`. To address that, Meta’s Folly queue [14] implements the partial, not total dequeue method, which spins until dequeue succeeds [33]. With such interface, dequeue calls return only if there is an entry to consume, otherwise blocking the thread indefinitely.

Another solution is to “fix the state” when `C.head` exceeds `P.head` by invalidating entries between them, as done by LCRQ [38]. Unfortunately, that causes consumers to hamper the progress of producers. SCQ [40] solves the producer starvation by limiting the number of consecutive invalidations with a threshold, as we describe below. Nevertheless, the remaining invalidations still degrade the enqueue performance, and the SCQ implementation [39] employs a trick

to reduce the probability of invalidating entries: Consumers check several times in a loop whether the entry has been committed before actually invalidating it. Unfortunately, this *delayed invalidation* trick increases the latency of dequeue when the queue is empty by several orders of magnitude — we experimentally demonstrate this empty-deq in Sec. 6.2.3.

(P2) Producer contention on P.head. A straightforward form of enq-enq interference is caused by multiple producers concurrently calling enqueue and contending on updates to P.head. Here, Folly queue again resorts to FAA and turns enqueue into a partial method, which waits until free entries are available, potentially blocking the thread forever.

Nikolaev proposes a novel idea to implement a total queue while using FAA [40]: SCQD combines two SCQ index queues (fq and aq) with a data array. Upon enqueue on SCQD, the thread gets an index from fq, copies the data in the corresponding entry of the data array, and then puts the index into aq. Dequeueing works the other way around. The index queues are total on dequeue but partial on enqueue, *i.e.*, dequeue returns EMPTY if the queue is empty, whereas enqueue loops until it succeeds. Nevertheless, the combined SCQD is still total since index queues are never full, *i.e.*, the number of indexes is fixed and matches the maximum size of the index queues. Besides the constant overhead introduced by index queues, the high latency caused by the empty-deq issue in each SCQ index queue translates into high latency in SCQD for both empty-deq and full-enq cases (see Sec. 6.2).

(P3) Delayed P.tail and C.tail updates. Another typical enq-enq or deq-deq interference arises from the *in-order* policy to commit (resp. consume) entries — the default policy in DPDK ringbuffer. The head/tail mechanism, which resembles a ticket lock, brings issues analogous to Lock-Holder- and Lock-Waiter-Preemption [45]. For example, the preemption of a thread that is about to update P.tail (resp. C.tail) causes other enqueue calls (resp. dequeue calls) to uselessly spin (Lines 10 and 23) for arbitrary time periods.

Several queues implement, instead, *out-of-order* policies, allowing producers (resp. consumers) to commit (resp. consume) entries independently. In LCRQ, once consumers increment C.head such that it reaches P.tail, they start invalidating entries until C.head reaches P.head. That prevents producers from committing entries at indexes preceding C.tail, ensuring linearizability [34]. This approach starves producers and even livelocks the queue. For example, a consumer in an ongoing dequeue invalidates an entry when $C.head = P.tail < P.head$; the producer in the ongoing enqueue increments P.head to retry; the consumer realizes C.head still did not reach P.head and retries consuming, potentially invalidating the new entry if not yet committed; and so on.

SCQ uses a threshold T to restrict the number of consecutive entries invalidated, and, thus, avoid livelocks. When a consumer invalidates an entry, it atomically decrements T . A successful enqueue resets T to its initial constant $3n - 1$, where n is twice the queue capacity. This constant is care-

fully derived to guarantee linearizability is never violated [40]. However, it introduces additional contention among producers and consumers updating the threshold variable, which has to be again mitigated by delayed invalidation.

DPDK [9] ringbuffer implements a more practical out-of-order policy called RTS mode [27], which trades linearizability to avoid invalidations. Consumers never move C.tail forward if C.tail would reach P.tail, returning EMPTY despite of any committed entry between P.tail and P.head; thus, violating linearizability. Producers employ the reciprocal strategy.

To enable out-of-order commits, RTS records whether all entries between P.tail and P.head are committed. The *prohibited window* between P.tail and P.head has dynamic length because P.tail is moved forward only once the last producer writing between P.tail and P.head commits. If producers would keep allocating entries, they would keep incrementing P.head and extending the prohibited window up to the total capacity of the queue. To prevent consumers from starving, RTS sets up a threshold to limit the maximum distance between P.tail and P.head. If that distance is reached, enqueue blocks until all producers between P.tail and P.head have committed. RTS enables out-of-order consumes with the reciprocal approach.

(P4) Causes of enq-deq interference. There are two sources of interference between enqueues and dequeues: algorithmic and cache-related. While focusing on enq-enq or deq-deq cases, previous techniques introduce algorithmic interferences between enqueue and dequeue, *e.g.*, requiring producers and consumers to retry operations, increasing latency, potentially causing thread starvation, or even livelock.

Let us again consider the simple algorithm of Fig. 1. Even though cache misses caused by writing or reading the data cannot be eliminated, cache misses on the control variables are relevant. Every time a producer calls enqueue, it allocates an entry and increments P.tail. Every time a consumer calls dequeue, it potentially suffers a cache miss by reading P.tail. If the producer is far ahead the consumer, the cache misses at P.tail seem unjustifiable. Similarly, the producer suffers cache misses on C.tail even when there is plenty space in the queue between producers and consumers. In contrast to enq-enq and deq-deq, enq-deq interference is relevant even to the single-consumer/single-producer scenario, an important scenario for the industry. In Sec. 6, we experimentally show a correlation between a strong decrease of L1 cache misses and the performance improvements of BBQ (Sec. 3 and 4).

3 Design of BBQ

3.1 The Block-based Approach

BBQ splits the ringbuffer into blocks, as shown in Fig. 2. Each block contains one or more entries, usually multiple, depending on the configuration. The queue control variables are also split into queue-level and block-level variables. Control variables C.head and P.head now point to blocks instead

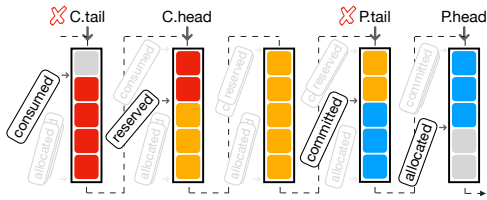


Figure 2: Block-based bounded queue (BBQ).

of entries; P.tail and C.tail are unnecessary for the algorithm. The block-level control variables include four cursors called allocated, committed, reserved, and consumed, which track the corresponding actions within each block.

The block-based approach greatly reduces the enq-deq interference. After a block is fully committed, its producer cursors (allocated/committed) remain unmodified until the block is fully consumed, causing no additional cache misses for consumers. Moreover, producers can always determine whether a block is fully allocated without accessing consumer cursors (reserved/consumed).

Multiple producers in the same block still contend on allocated and committed. Fortunately, the block-based approach enables the enqueue operation to use FAA, avoids costly invalidations, and still allows for a total method. Producers start using a block only once it has been fully consumed. Therefore, inside the block, FAA never allocates an entry that is not consumed yet, allowing enqueue to be total. FAA may make allocated out of the bound of the block, but state fixing is not required. Since each block has its own control variables, an out-of-bound cursor in one block does not affect the following block.

Although our dequeue operation employs CAS to avoid consumers from invalidating entries currently used by producers, BBQ still achieves similar or better performance than other designs with FAA-based dequeues. To further improve performance for machines with Armv8.1 [2] processors supporting Large System Extensions [3] (LSE), BBQ uses the atomic maximum instruction MAX instead of CAS in dequeue.

Finally, the block-based approach enables a practical out-of-order policy similar to RTS mode of DPDK. In this case, instead of updating control variables with double-width CAS, BBQ employs more scalable FAA and MAX instructions.

3.2 BBQ from a Bird's-eye View

In this section, we describe the high-level algorithm of BBQ, as shown in Fig. 3.

Producers. To enqueue data, a producer first retrieves the current value of P.head and the corresponding block identifier (Line 4). Next, it tries allocating an entry in the block (Line 5). If successful, the producer writes the data into entry ety and commits it (Line 7). The allocation fails if the block has already been fully allocated (Line 9). In this case, the

```

1  status := OK(T) | FULL | EMPTY | BUSY
2  status BBQ<T>::enqueue(T data){
3  loop:
4      (ph, blk) = get_phead_and_block();
5      switch (allocate_entry(blk)){
6          case allocated(ety):
7              commit_entry(ety, data);
8              return OK();
9          case BLOCK_DONE:
10             switch (advance_phead(ph)){
11                 case NO_ENTRY: return FULL;
12                 case NOT_AVAILABLE: return BUSY;
13                 case SUCCESS: goto loop;
14             }
15         }
16     }
17 status BBQ<T>::dequeue(){
18 loop:
19     (ch, blk) = get_thead_and_block();
20     switch (reserve_entry(blk)){
21         case reserved(ety):
22             data = consume_entry(ety);
23             if (data != NULL) return OK(data);
24             else goto loop;
25         case NO_ENTRY: return EMPTY;
26         case NOT_AVAILABLE: return BUSY;
27         case BLOCK_DONE(vsn):
28             if (advance_thead(ch, vsn)) goto loop;
29             else return EMPTY;
30     }
31 }

```

Figure 3: High-level design of BBQ.

producer tries to advance P.head to the next block (Line 10). If successful, the producer jumps back to the loop label and retries the allocation (Line 13). In retry-new mode, advancing P.head fails if the next block is not yet fully consumed, *i.e.*, the whole queue is considered full. BBQ distinguishes the failure reason: BUSY when some dequeue operation is ongoing and FULL otherwise. Returning BUSY allows for custom back-off implementations at the caller side, *e.g.*, parking threads after a number of retries. In drop-old mode, advancing P.head does not fail except for a seldom case discussed in Sec. 4.3, for which BUSY is returned.

Consumers. The dequeue operation is somewhat analogous to enqueue. The consumer starts by retrieving the current value of C.head and the corresponding block identifier. Next, it attempts to reserve an entry to consume (Line 20), advancing reserved. If the reservation succeeds, the consumer reads the data (Line 22) and advances consumed. In drop-old mode, the consumer may have to retry consuming if the producers have overwritten the block (Line 24). Reserving an entry can fail in several ways. When the next entry in blk is allocated but not yet committed, dequeue returns BUSY (Line 26). When blk is not fully allocated and all committed entries were already consumed, dequeue returns EMPTY (Line 25). Finally, when blk is fully committed and fully consumed, the consumer tries advancing C.head (Line 28). Upon success, it retries reserving an entry, jumping to loop. Otherwise, dequeue returns EMPTY.

Progress guarantees. Similarly to DPDK ringbuffer, BBQ is a deadlock-free queue, its progress depend on a fair scheduler. In contrast to DPDK, BBQ is less affected by CPU oversubscription (see Figure 9g, Section 6). To see why this is the case, consider the situation where DPDK producers form a waiting chain: the last to allocate an entry can only commit once the previous has committed its entry, and so on. This waiting chain hampers the performance because the scheduler is unlikely to unpark the preempted producers in the chain's order. In BBQ, there is no such waiting chain, *i.e.*, producers commit independently. Consumers may wait for producers only in seldom cases. For example, a consumer waits for a preempted producer on the same block if the producer has allocated but not yet committed an entry. Nevertheless, any order in which the fair scheduler unparks preempted producers allows the consumer to make progress.

3.3 Two-Level Control Variables

Essentially, BBQ splits the control variables into two levels, namely the queue-level and block-level variables (see Fig. 2). Queue-level control variables point to blocks (C.head and P.head), whereas block-level control variables to entries (allocated, committed, reserved, and consumed).

Versions. As in other queues such as DPDK ringbuffer, control variables have to be versioned to identify multiple reuses of the same memory locations and, in this way, avoid ABA problems¹. Therefore, queue-level control variables have two fields, an *index* pointing to a block and a *version* identifying how many rounds the whole queue has been reused. Similarly, block-level control variables have an *offset* field pointing to an entry within the block and a *version* field identifying how many times the block has been reused.

Phantom heads. Before producers can allocate entries in a block *B*, one producer has to reset *B*'s allocated cursor as well as advance P.head to point to *B*. Without making both updates atomic, whichever update executes first may trigger an ABA problem as well. To allow both being updated atomically, we introduce the concept of *phantom head*, which is based on the following observation. The index and version values of P.head can be inferred from the versions of all allocated cursors in the queue (as described in Sec. 4.2.2). Similarly, the phantom C.head can be inferred from the versions of all reserved cursors. Since the phantom P.head (resp. phantom C.head) is implicitly updated whenever any allocated cursor (resp. reserved cursor) is updated, we use them instead of queue-level head variables.

¹Often algorithms try to guarantee operation atomicity by reading from a control variable before and after the operation. If the same value *A* is read both times, the programmer assumes absence of concurrent updates and hence that the operation was atomic. This assumption breaks if other threads can temporarily change the value to $B \neq A$ and then back to *A*; algorithms in which this situation can occur are said to suffer from the ABA problem [32].

Cached heads. In principle, phantom heads allow us to eliminate the C.head and P.head variables altogether. Unfortunately, phantom heads are costly: To infer them, one needs to compare the cursors of every block. Instead of eliminating C.head and P.head, we consider them to be *cached heads*, *i.e.*, potentially stale values of the phantom heads. Cached heads only exist for performance reasons; their staleness does not affect correctness.

4 Implementation of BBQ

Figure 4 shows the low-level detail of BBQ, including data-fields, enqueue and dequeue operations for retry-new mode and drop-old mode. The drop-old mode will be introduced in Sec. 4.3.

4.1 Structure

Heads and cursors. BBQ has two queue-level Head variables and four block-level Cursor variables in each block. Head and Cursor types are 64-bit integers, which can be atomically updated. We reserve two bit-segments in Head to represent the version and index and two bit-segments in Cursor to represent the version and offset. Given a total number of blocks (BLOCK_NUM) and the capacity of a block (BLOCK_SIZE), the segments have the following bit-lengths:

$$\begin{aligned} |\text{Index}| &= \log_2(\text{BLOCK_NUM}) \text{ bits} \\ |\text{Offset}| &> \log_2(\text{BLOCK_SIZE}) \text{ bits} \\ |\text{Version}| &= 64 - \max(|\text{Index}|, |\text{Offset}|) \text{ bits} \end{aligned}$$

The bit-length of Offset is larger than $\log_2(\text{BLOCK_SIZE})$ to allow for FAA-overflow detection. The Index and Offset are the least significant bits of Head and Cursor, respectively; Version bits immediately follow them; and reminder bits, if existent, are set to 0 and ignored. That allows us to easily manipulate these fields with FAA and MAX instructions.

For convenience, we access the bit-segments from Head and Cursor variables as if they were regular fields named idx, off, and vsn, *e.g.*, allocated.idx. Moreover, we construct variables (*e.g.*, Head) with the short-hand notation Head{.vsn=version, .idx=index}, initializing unspecified fields with 0. We may omit the type when clear from the context.

Initially, idx and off in the first block are zero and for remaining blocks off is set to BLOCK_SIZE. The initial value of vsn will be introduced in Sec. 4.2.2.

Other types. Block has shared cursors, annotated with shared<>, and an array of entries of type T (Line 38). EntryDesc is an entry descriptor; it points to a block and contains offset to location the actual entry and a version data-consistency checks used in drop-old mode (Line 41). Finally, BBQ contains the shared heads and an array of Block<T>.

```

1 <Head, Block> BBQ<T>::get_phead_and_block(){
2   ph = LOAD(phead);
3   return (ph, blocks[ph.idx]);
4 }
5 state BBQ<T>::allocate_entry(Block blk){
6   if (LOAD(blk.allocated).off >= BLOCK_SIZE)
7     return BLOCK_DONE;
8   old = FAA(blk.allocated, 1).off;
9   if (old >= BLOCK_SIZE)
10    return BLOCK_DONE;
11  return ALLOCATED(EntryDesc{.block=blk, .offset=old});
12 }
13 void BBQ<T>::commit_entry(EntryDesc e, T data){
14   e.block.entries[e.offset] = data;
15   ADD(e.block.committed, 1);
16 }
17 state BBQ<T>::advance_phead(Head ph) {
18   nblk = blocks[(ph.idx + 1) % BLOCK_NUM];
19   cons = LOAD(nblk.consumed);
20   if (cons.vsn < ph.vsn ||
21       (cons.vsn == ph.vsn && cons.off != BLOCK_SIZE)) {
22     reserved = LOAD(nblk.reserved);
23     if (reserved.off == cons.off) return NO_ENTRY;
24     else return NOT_AVAILABLE;
25   }
26   cmtd = LOAD(nblk.committed);
27   if (cmtd.vsn == ph.vsn && cmtd.off != BLOCK_SIZE)
28     return NOT_AVAILABLE;
29   MAX(nblk.committed, Cursor{.vsn=ph.vsn + 1});
30   MAX(nblk.allocated, Cursor{.vsn=ph.vsn + 1});
31   MAX(phead, ph + 1);
32   return SUCCESS;
33 }
34 class BBQ<T> {
35   shared<Head> phead, chead;
36   Block<T>[] blocks;
37 }
38 class Block<T> {
39   shared<Cursor> allocated, committed;
40   shared<Cursor> reserved, consumed;
41   T[] entries;
42 }
43 class EntryDesc {
44   Block block; Offset offset; Version version; }

```

```

45 <Head, Block> BBQ<T>::get_thead_and_block(){
46   ch = LOAD(thead);
47   return (ch, blocks[ch.idx]);
48 }
49 state BBQ<T>::reserve_entry(Block blk){
50   again:
51   reserved = LOAD(blk.reserved);
52   if (reserved.off < BLOCK_SIZE) {
53     committed = LOAD(blk.committed);
54     if (reserved.off == committed.off)
55       return NO_ENTRY;
56     if (committed.off != BLOCK_SIZE){
57       allocated = LOAD(blk.allocated);
58       if (allocated.off != committed.off)
59         return NOT_AVAILABLE;
60     }
61     if (MAX(blk.reserved, reserved + 1) == reserved)
62       return RESERVED((EntryDesc){.block=blk,
63                                     .offset=reserved.off, .version=reserved.vsn});
64     else goto again;
65   }
66   return BLOCK_DONE(reserved.vsn);
67 }
68 T BBQ<T>::consume_entry(EntryDesc e){
69   data = e.block.entries[e.offset];
70   ADD(e.block.consumed, 1);
71   allocated = LOAD(e.block.allocated);
72   if (allocated.vsn != e.version) return NULL;
73   return data;
74 }
75 bool BBQ<T>::advance_thead(Head ch, Version vsn){
76   nblk = blocks[(ch.idx + 1) % BLOCK_NUM];
77   committed = LOAD(nblk.committed);
78   if (committed.vsn != ch.vsn + 1)
79     return false;
80   MAX(nblk.consumed, Cursor{.vsn=ch.vsn + 1});
81   MAX(nblk.reserved, Cursor{.vsn=ch.vsn + 1});
82   if (committed.vsn < vsn + (ch.idx == 0))
83     return false;
84   MAX(nblk.reserved, Cursor{.vsn=committed.vsn});
85   MAX(thead, ch + 1);
86   return true;
87 }

```

retry-new mode | drop-old mode

Figure 4: Low-level details of BBQ.

4.2 Operations

Enqueue and dequeue operations are divided into different cases: First, when the allocation in the enqueue or the reservation in the dequeue do not fail. Second, when enqueue or dequeue have to advance respective heads to the next block.

4.2.1 Successful allocation/reservation

The producer uses `FAA` to allocate an entry (Line 8) and returns its location as `EntryDesc` if there is enough space in the current block (Line 11). A pre-check (Line 6) avoids endless increasing of `allocated` when the queue is full, which could cause `FAA` overflows and impact performance negatively. For the consumer, the entry is reserved through `MAX`² (Line 61),

²Unlike `FAA`, `MAX` provides conditional update semantics. Moreover, for some cases, `MAX` has similar semantics to `CAS` but better performance observed

which atomically sets a variable if the given value is greater than the variable's value and returns the old value. Consumers never pass producers (Line 54) and can read when out-of-order commit are not ongoing in the same block, which means all allocated entries are committed (Line 58).

4.2.2 Advancing to the next block

Monotonic version updates. Head and cursor versions are initially zero. Both enqueue and dequeue calls start by reading the current cached head (`phead` and `thead`, respectively) into a local variable (`ph` and `ch` in Fig. 3). After failing to allocate or reserve an entry, these calls try to advance the respective phantom heads by calling `advance_phead` or `advance_thead`.

from experimental results. We use `CAS` and while loop to achieve the same functionality for architectures that do not support `MAX` such as x86 [16].

These functions *try* to reset the cursors of the next block with the previously read version of the cached head plus one (Lines 29, 30, 80, and 81 in Fig. 4). Subsequently, the functions *try* update the cached head itself (Lines 31 and 85).

The reset of cursors and the update of cached head may not always succeed. Consider the following example. Two producers try to allocate entries at block B_0 and fail. Both have read `phead` with value `{.vsn=0, .idx=0}`. Now both call `advance_phead` concurrently and are at Line 30. Producer P_1 stalls while producer P_2 succeeds updating the allocated cursor of block B_1 . P_2 also allocates one or more entries such that now B_1 's `allocated` has the value `{.vsn=1, .off=16}`. If now P_1 would be able to succeed resetting `allocated`, then the allocations of P_2 would be lost. Nevertheless, to avoid such ABA situations, the reset of cursors and update of cached head do not have to be performed with an expensive CAS. The recent MAX atomic instruction from Armv8.1-LSE can provide the required monotonicity.

Invariants. Producers have to ensure they advance `phead` only if the next block that has no unconsumed data. Consumers have to ensure they advance `chead` only if the next block has committed data.

We guarantee these invariants by ensuring that the version difference between phantom `phead` and phantom `chead` never exceeds 1. When producers advance `phead` and reset the `allocated` and `committed` cursors of the next block with version `ph.vsn+1` (Line 30), the `consumed` cursor must have version `ph.vsn` (Line 21). Similarly, when consumers advance `chead` and reset the `reserved` and `consumed` cursors of the next block with version `ch.vsn+1` (Line 80), the `committed` cursor must have version `ch.vsn+1` (Line 79).

Order matters. Often the order in which shared variables are accessed is crucial for correctness. For example, reading `reserved`, `committed`, and `allocated` variables (Lines 51, 53, and 57) in a different order can cause the consumer to read garbage. Moreover, updating cached heads (Lines 31 and 85) must happen after updating block-level variables (Lines 29, 30, 80, and 81), otherwise blocks may be fully skipped.

To guarantee shared variables are accessed in the program order of Fig. 4 on architectures with weak memory models, C/C++ implementations of BBQ can employ atomic LOAD, STORE, MAX, FAA, and CAS instructions with sequentially consistent memory barriers (see C11/C++11 atomics [6]). In Sec. 5, we report a correct relaxation of these barriers.

4.3 Drop-old Mode

Unlike the retry-new mode, where producers cannot insert data when the queue is full, in drop-old mode, producers continue to write even if the data is not yet consumed. Consequently, producers no longer depend on the consumers to make progress. The FIFO property still holds, except that some data might be lost. In other words, entries are consumed

in the order in which they were allocated, but some committed entries may not be consumed.

Speculative reads. Drop-old mode is widely used in profiling scenarios, where enqueue calls writing a log should not be delayed by dequeue calls that read the log. To reduce the chances of dequeue calls interfering with enqueue calls, consumers read data in a speculative fashion. The consumer first reads the data and then checks whether it has been overwritten. If so, it discards the data and tries reserving another entry.

From retry-new to drop-old mode. A few differences exist between retry-new and drop-old mode. First, producers avoid advancing to blocks that are still not fully committed in the previous round, returning BUSY (Lines 27 and 28).

Second, consumers guarantee FIFO order by checking if the version of the next block is greater than or equal to the current one (Line 82). If that is the case, `reserved` is reset with the version of `committed` (Line 84), indicating the block is ready to be read. The first block is a special case because, in contrast to other blocks, its version is always off-by-one. Therefore, we add 1 to the comparison if `ch.idx == 0`.

Third, the data-consistency check is based on the fact that a block is not overwritten as long as `allocated` and `reserved` versions are equal. Therefore, before reading data, we record the `reserved` version (Line 63), and after copying the data from the entry, we check if corresponding `allocated` version still matches the `reserved` (Line 72).

4.4 Variable-sized Entries

BBQ can support variable-sized entries with minor algorithmic changes. Each entry has an additional metadata `size` to support different entry sizes in one queue. Block-local cursors and `BLOCK_SIZE` indicate the space of entries instead of their number. MAX at Line 61 is no longer sufficient; CAS must be used instead.

Dummy entry. Unlike the fixed-size version of BBQ, where entries can exactly fill up a block, here, the remaining space of a block might not be enough to contain the new entry. In such cases, we mark the space with a dummy entry and return `BLOCK_DONE` to trigger a retry in the next block. Since enqueue uses FAA, the producers that cause `allocated` go over the boundary marks the dummy entry by setting its `size` to zero and commits it. Consumers that read an entry with `size` zero ignore the dummy entry and retry in the next block. Upon reading the dummy entry, the consumer also sets `consumed` to be equal to `BLOCK_SIZE`.

4.5 Other Implementation Details

We have implemented BBQ in C and Java. We have also implemented a wrapper with the Java Native Interface (JNI) [20] to call the C version from Java.

Finally, we have optimized BBQ for SPSC scenarios as follows: (1) `phead` and `chead` are no longer shared variables and

can be accessed with non-atomic loads/stores. (2) `allocated` and `committed` (resp. `reserved` and `consumers`) are merged into one variable and updated with `STORE`.

5 Verification and Optimization of BBQ

Concurrent data structures are complicated beasts and are easy to get wrong [30]. To increase confidence in our C implementation and find intricate bugs, we generate a series of small hand-crafted tests that can trigger corner cases in the algorithm and then use `VSYNC` [42], an extension to the GenMC model checker [36]. The tool generates all executions of the algorithm on those tests, including executions that can only happen on WMMs, exercising the following critical corner cases: (1) queue full or empty, (2) FIFO, (3) wrap-around, and (4) termination of bounded loops with bounded effect [37, 42].

Bugs. We found three concurrency bugs in an earlier version of the drop-old mode of BBQ.

1. A test revealed a bug in which enqueue operations incorrectly returned `BUSY`. The block was detected as `NOT_AVAILABLE` because, in that version, the condition at Line 27 was `committed.off == BLOCK_SIZE && committed.vsn == ph.vsn`. Therefore, even if other producers reset the next block and have the space to allocate, the block would still be `NOT_AVAILABLE`. That violated linearizability.
2. We have found a termination bug in which the checking in Line 82 was written as `blk.committed.vsn >= nblk.committed.vsn`, missing the special case of the version number in the first block, which may let consumers advancing the block forever if the queue is empty and all blocks happen to have the same version number.
3. The wrap-around test revealed a bug due to a missing fence, where readers could return incorrect data when a fast writer overwrote the entry they were currently reading.

We found these bugs through the verification with model checking. They were not found during stress testing, nor by running the small test cases directly on hardware. However, we could retrospectively construct test cases that reproduce these bugs on real hardware. Concurrent algorithms, especially those using complicated synchronization such as drop-old mode, are hard to get right using traditional methods.

Barrier optimization. We used `VSYNC` to run the memory barrier optimization for WMMs. The results consistent with the order analyze of reading/updating shared variables in Sec. 4.2.2. For the fixed entry size version, 14 atomic instructions with full memory barriers are optimized to 3 release barriers, 3 acquire barriers, and 8 relaxed barriers, respectively.

6 Evaluation

6.1 Environment Setup

Hardware. All of our experiments are performed on three x86 machines with 88, 96, and 12 hyperthreads, respectively (denoted as x86-88T, x86-96T, and x86-12T), and an ARM machine with 96 cores (arm-96T). x86-88T and x86-96T are connected through two 10Gbps links.

Software. On these servers, we installed Ubuntu 20.04.3 LTS, with Linux kernel 5.4.0. We use Linux `perf` [26] to get results of L1 cache misses, the version of it is the same with the Linux kernel. Java-based experiments use JDK v11 [19].

6.2 Microbenchmarks

Workloads. We have the following 3 workloads for microbenchmarks implemented in C/C++ and Java:

- **simple:** Each producer or consumer has its own thread, where they keep executing enqueue or dequeue operations in a loop. Data is validated after each dequeue.
- **complex:** Based on the simple workload. Producers and consumers allocate space for data, preform enqueue and dequeue then manually free (C/C++ version) or let JVM garbage collection it [46] (Java version). Additionally, each operation also performs a deterministic random busy-loop of at most one hundred `nop` instructions.
- **profiling:** Based on the simple workload. The throughput of producers and consumers is fixed at 10kop/s and 1kop/s, respectively.

Thread affinity. For MPSC or SPMC scenarios, we assign a single producer or consumer at the first core/hyperthread and then distribute the other threads sequentially to cores/hyperthreads. For MPMC, we assign producers and consumers interleaved one by one; if their number differs, the surplus is assigned at the end.

Experiments. We perform the following experiments, each measuring a different metric:

- **throughput:** Total number of consumed entries per second.
- **data-latency:** Average time each data stays in the queue.
- **op-latency:** Average latency of each enqueue or dequeue operation.
- **cache-miss:** Average number of L1 cache misses per consumed entry, measured with Linux `perf`.
- **fairness:** Throughput of each producer and consumer (only for MPSC and SPMC).
- **full/empty:** Latency of enqueue when the queue becomes full and latency of dequeue when the queue becomes empty (only used with simple workload).
- **oversubscription:** Throughput with more producers and consumers than than cores/hyperthreads.

Each experiment runs 3 times. If not specified otherwise, solid lines represent average results; shaded area represents

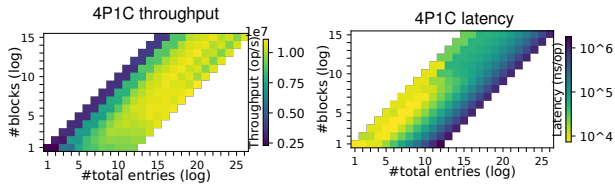


Figure 5: BBQ throughput and latency varying number of blocks and entries (x86-88T).

standard deviation; and vertical dashed lines indicate when threads cross NUMA nodes, are assigned to hyperthreads in the same core, or are oversubscribed.

Configuration. The data size is always 8 bytes, a size all queues can support. For the data-latency experiment, the number of entries is around 128. For the other experiments, the buffer size is 32k bytes unless specified otherwise.

6.2.1 BBQ Parameters and Design Choices

We start by evaluating parameters and design choices of BBQ.

Configuring the number of blocks. Figure 5 shows throughput and data-latency experiments for BBQ with four producers and one consumer. The color scale shows the existing trade-off between number of entries and number of blocks; users have to be aware of that when choosing the buffer size and number of blocks. We use the following heuristic function to determine the number of blocks in all rest experiments: $number\ of\ blocks\ (log) = \max(1, \lfloor number\ of\ entries\ (log) / 4 \rfloor)$.

Performance impact of FAA. Figure 6 shows the results of an MPSC throughput experiment on our Arm machine. BBQ is configured to use FAA instruction from Armv8.1 LSE, standard FAA and CAS implemented with load-exclusive and store-exclusive instructions. Except for the 1 thread case, LSE-based FAA shows the best scalability, outperforming the other two by at least 5 times.

Support for variable-sized entries. Figure 6 also shows the throughput of the BBQ with fixed- and variable-sized entries. The size of each data is the same, yet the varied entry version has to store additional size information for every entry. Nevertheless, the throughput difference between both is negligible.

Consumer-producer interference in drop-old mode. Finally, Figure 6 shows the throughput of BBQ with drop-old mode in two configurations: MPSC and MPNC (multiple-producer/no-consumer). The throughput of the producers (nr-prod) with no consumers is less than 8% higher than with consumer (sr-prod). Moreover, the consumers manage to consume at least 99.97% of the entries except for the 1 thread case (sr-cons). These results illustrate that consumers with the speculative-read method incur a rather minor interference on producers — please refer to Sec. 6.2.3 for a baseline with existing implementations.

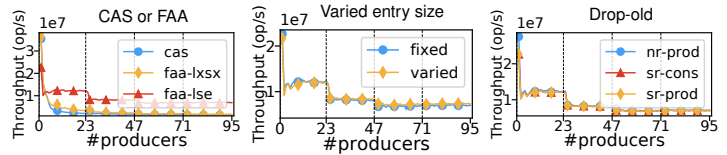


Figure 6: BBQ throughput with CAS and FAA; with support for variable-sized entries; and with drop-old mode (arm-96T).

6.2.2 State-of-the-art Comparison: Retry-new Mode

We now compare BBQ against 5 state-of-the-art bounded queues: dpdkrb, DPDK ringbuffer v21.08 [9]; scqd, a lock-free bounded queue [40]; linuxrb, the ringbuffer in the Linux kernel v5.16 [22]; boostq, the bounded queue in C++ Boost libraries v1.71 [4]; and follyq, the bounded queue (with total method) [25] in Meta’s open-source Folly library v2021.11.8. For dpdkrb and follyq, we use their SPSC versions to run corresponding SPSC experiments.

Effectiveness of the Block-based Approach. To isolate the effect of the blocks, we first focus on SPSC experiments because BBQ do not profit from FAA in such scenarios. Figure 7 shows that BBQ greatly outperforms all other bounded queues in all experiments. For the simple workload, BBQ yields 11.3x to 42.4x higher throughput than other libraries. The throughput of BBQ is $1.41 \cdot 10^8$ op/s, while the second-best one follyq is $1.24 \cdot 10^7$ op/s. For the complex workload, which has a random busy-loop to limit the maximum throughput, BBQ still outperforms follyq by 2x. BBQ’s better performance is mainly due to the massive decrease in L1 cache misses with the block-based approach (notice the y-axis log scale).

Throughput in MPSC and SPMC scenarios. Figures 9a and 9b show BBQ performing on par or better than other queues in the simple and complex workloads. For MPSC scenarios, BBQ performs up to 10.13x and 3.65x faster than the second-best queue, respectively. For SPMC scenarios, BBQ performs up to 1.88x and 2.39x faster than the second-best queue, respectively.

The throughput difference between MPSC and SPMC results can be attributed in part to the different L1 cache misses measurements (see Fig. 9c). BBQ consumers employ CAS operations in every dequeue, and these can fail and have to be retried, each time suffering another cache miss.

Data latency. We measure the average time data stays in the queue in the complex workload, as shown in Fig. 9d. For MPSC case, BBQ performs consistently better than other bounded queues; up to 17.22x lower latency than the second-best queue. For the SPMC case, scqd performs best, up to 7.45x lower latency than BBQ. That is an artifact of the delayed invalidation trick (see Sec. 2): Once the queue is empty (C.head = P.tail), consumers invalidate the entries pointed by C.head after a delay. Since consumers first increment C.head and then wait, multiple consumers will be pending on differ-

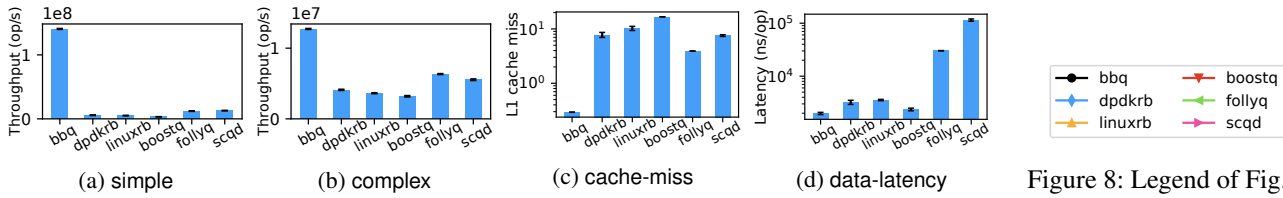


Figure 7: SPSC comparison of BBQ against state-of-the-art on x86-88T.

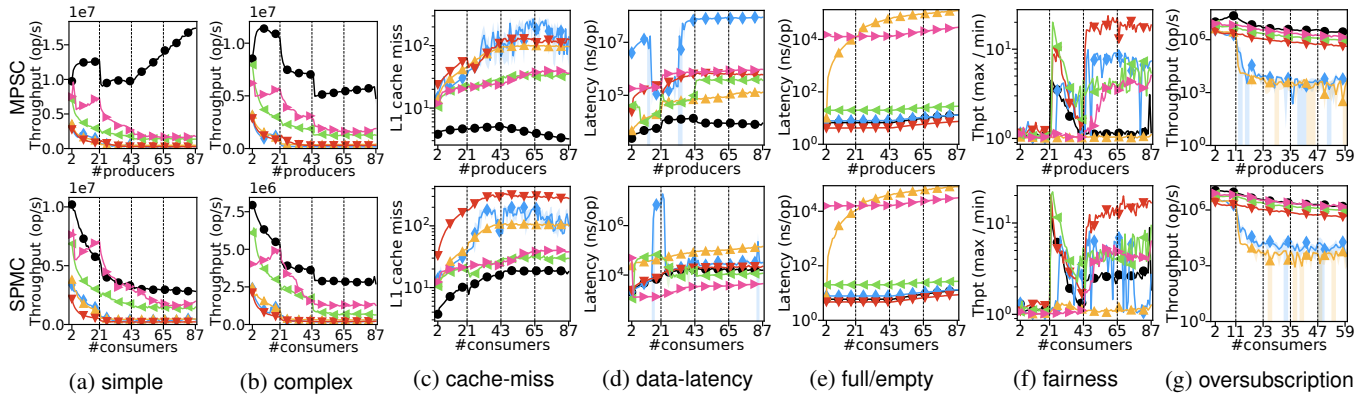


Figure 9: MPSC and SPMC comparison of BBQ against state-of-the-art on x86-88T (and x86-12T for oversubscription).

ent entries. As soon as the producer commits a new entry, one consumer aborts its delay and immediately returns the data.

Full and empty queues. Figure 9e shows the latency for failed enqueue on a full queue (top figure), and failed dequeue on an empty queue (bottom figure). In such scenarios, the delay invalidation of scqd incurs a high cost: the latency of failed operations in scqd is around 1000x higher than in most other queues. For linuxrb, the latency increases with the number of producers/consumers due to its coarse-grained locking.

Fairness between producers or consumers. Figure 9f shows the relation between maximum and minimum throughput of producers (top figure) and consumers (bottom figure). linuxrb provides exceptional fairness because it relies on a fair spinlock³. Other queues show unfair throughput after crossing the first NUMA node (at 22 producers/consumers) except for scqd, which becomes unfair when hyperthreads of the same cores start being used (at 44 producers/consumers).

Oversubscription effects. Figure 9g shows the results of our oversubscription experiment on x86-12T with up to 5x more threads than hyperthreads. Both dpdkrb and linuxrb are highly affected by oversubscription; the former due to their in-order policy (see Sec. 2), the latter due to its coarse-grained locking. Under oversubscription (*i.e.*, with more than 12 threads), BBQ outperforms the second-best queue by a small margin: 2.22x in MPSC and 1.23x in SPMC scenarios.

³In our userspace port of linuxrb, we employ a ticket lock.

6.2.3 State-of-the-art Comparison: Drop-old Mode

We now compare BBQ with other two bounded queues that support overwriting old values, namely EvictingQueue from Google Core Libraries Guava [15], and CircularFifoQueue from Apache Commons [1]. The experiments are conducted on the arm-96T machine.

Producer performance. Figure 10a shows the enqueue throughput with *no consumers* for the complex workload. On the one hand, BBQ-JNI yields 3.2x higher enqueue throughput than EvictingQueue and CircularFifoQueue. On the other hand, BBQ yields an enqueue throughput rather similar to them. Intuitively, BBQ-JNI has a better performance since employs real *FAA* instructions, whereas, in the Java version of BBQ, the JVM translates *FAA* into *CAS* [35].

Figure 10c shows the enqueue latency, again with *no consumers*, for the profiling workload. Remember that producers issue 10k enqueue calls per second in the profile workload. With BBQ and BBQ-JNI, the enqueue latency slowly increases: 147.9ns and 176.4ns with 1 thread, respectively, to 965.6ns and 914.3ns with 94 threads, respectively. Up to 44 producers, EvictingQueue and CircularFifoQueue perform similar to BBQ variants. With more than 44 producers, however, their enqueue latency quickly increases up to 70 μ s (72x higher than BBQ). From Fig. 10a, we know that their maximum enqueue throughput is about 450kops. Hence, these queues already reached throughput limit with 44 producers, and any additional producers can only increase the latency. We believe the spike at 95 threads (BBQ with 5.1 μ s and BBQ-JNI with 2.0 μ s)

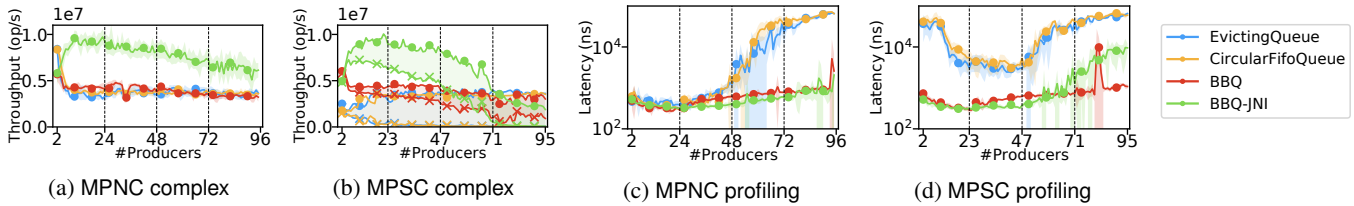


Figure 10: Cross comparison results for drop old mode on arm-96T.

may caused by garbage collection, but further investigation is necessary.

Enq-deq interference. We now introduce a single consumer to understand the interference of dequeue on the enqueue operations. Ideally, the enqueue operations should incur a small overhead (latency) to the profiled program; and this overhead should be minimally affected by concurrent dequeue calls. Moreover, if enqueue calls interfere with dequeue calls too frequently, more data may be dropped, *i.e.*, overwritten before being consumed.

Figure 10b shows the enqueue and dequeue throughput (marked with • and ×, respectively) for the complex workload. Comparing Figs. 10a and 10b, we observe that the enqueue throughput of BBQ is similar in both figures and of BBQ-JNI is similar up to 47 threads, but after that it drops to about $1.89 \cdot 10^6$ op/s. The enqueue throughput of EvictingQueue and CircularFifoQueue is initially lower when a consumer is concurrently calling dequeue. The reason for this lower enqueue throughput can be explained by observing the difference between enqueue and dequeue in Fig. 10b.

First, note that with more than a few producers, the enqueue and dequeue throughput of each queue do not match, *i.e.*, the consumer is not fast enough to read out all the data before the producers start overwriting the oldest entries. Also note that the more the enqueue throughput of EvictingQueue and CircularFifoQueue recovers (by increasing producers), the lower is the dequeue throughput. Once their throughput is back to the level of Fig. 10a with 15 threads, their dequeue throughput is no more than $4.92 \cdot 10^5$ op/s. In contrast, BBQ and BBQ-JNI sustain a much higher dequeue throughput up to 46 threads ($2.89 \cdot 10^6$ op/s and $5.07 \cdot 10^6$ op/s, respectively).

Figure 10d shows the enqueue latency for the profiling workload. BBQ and BBQ-JNI provide enqueue latencies varying from 730.6ns and 519.9ns with 2 producers, respectively, up to 1082ns and 9503ns with 95 producers — we ignore the noisy region with 81 producers. Comparing the results of Figure 10c and Figure 10d reveal that the enqueue latency of EvictingQueue and CircularFifoQueue, for example with 8 producers increase by 124.97 times when adding a single consumer with a relatively low dequeue frequency.

The latency increases as well as the throughput decreases of BBQ-JNI after 47 producers could be related to the JNI overhead of calling C code from Java.

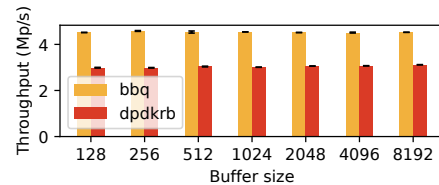


Figure 11: Throughput comparison between BBQ or DPDK ring buffer.

6.3 Macrobenchmarks

We now explore three benchmarks that represent the real-world usage queues.

6.3.1 DPDK’s End-to-end Benchmark

We replace the ring buffer in DPDK’s event library [12] and network driver [11] with BBQ, and run the multiprocess benchmark [8] from the DPDK Test Suite [10] (DTS). The benchmarks consists of one server process receiving and distributing packets, and two client processes performing level-2 packet forwarding [7]. These processes run on the device under test (DUT), our x86-88T machine. The tester and traffic generator TRex [29] run on our x86-96T machine. The packet size is 64 bytes (along with the UDP header) as well as the entry size of the queue. The versions of DPDK, DTS, and TRex are 21.08, 21.02, and 2.92, respectively. We report the end-to-end throughput (in million packets per second) measured by the traffic generator.

Figure 11 shows our experimental results. BBQ provides 1.5x higher throughput with different buffer sizes in the driver. We observed no further improvements with larger buffer sizes, indicating that the ring buffer may not be a bottleneck any more. We also replaced the so-called software queue in the multiprocess benchmark, and observed no improvement.

6.3.2 Linux io_uring

Linux io_uring [13] is a new asynchronous I/O [31] API for kernel-user space communication. It consists of two ring buffers, one for request submissions (SQ) and another for completion confirmations (CQ). It supports batched submission and batched confirmations with configurable batch sizes [18].

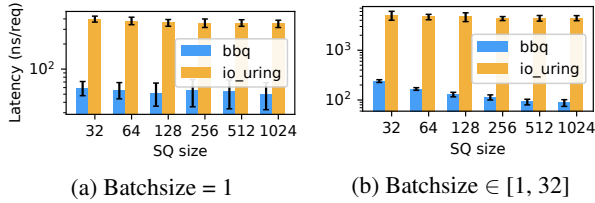


Figure 12: Latency per request comparison of BBQ and Linux `io_uring` on x86-88T.

We port `io_uring` from Linux kernel (v5.14-rc6) [17] to userspace, omitting I/O related functionality and replacing its ring buffers with BBQ. To avoid unstable results, we disable the option of overflowing entries into an additional linked list. We set the CQ size to twice the SQ size as recommended [21]. Our benchmark runs three threads: The first submits request batches (via SQ); the second (representing the kernel) consumes them and immediately produces confirmations (via CQ); and the third consumes the confirmation batches. We configure submission and confirmation batches with size 1 or with a random value from 1 to 32. Each experiment runs 10 times, measuring the time to submit 1M requests.

Figure 12 shows a significant improvement of the latency per request when using BBQ. For example, with batch size of 1 and SQ size of 32 and 1024, BBQ yields 6.7x and 6.9x lower latency than the original ring buffer, respectively. For random batch size and the same SQ sizes, BBQ yields even lower latencies: 20.9x and 50.5x, respectively.

6.3.3 LMAX Disruptor Benchmarks

Disruptor [23] is concurrency mechanism used for high-performance financial exchange. Its core component is a ring buffer. We compare its throughput with the Java and JNI versions of BBQ with three official Disruptor benchmarks: `OneToOneThroughputTest`, `ThreeToOneThroughputTest`, and `OneToThreeThroughputTest`. We modify these benchmarks to support not just three, but more producers or consumers. Apart from this modification, all other parameters (e.g., number of iterations, sleep time between operations, number of repetitions) are unchanged.

Disruptor can randomly change the batch size based on the workload. To make the comparison as fair as possible, we first run the benchmark with Disruptor to get the average batch size used, and then run BBQ with that batch size. Figure 13 shows the throughput of Disruptor, BBQ, and the baseline Java queue (`java.util.concurrent.BlockingQueue`) for several scenarios. The number on each bar refers to the (average) batch size, and the label $pPcC$ indicates the number of producers (p) and consumers (c).

In the 1P1C scenario, Disruptor yields almost 3x higher throughput than the Java queue (3 Mop/s versus 1.3 Mop/s). BBQ and BBQ-JNI, however, yield an order of magnitude

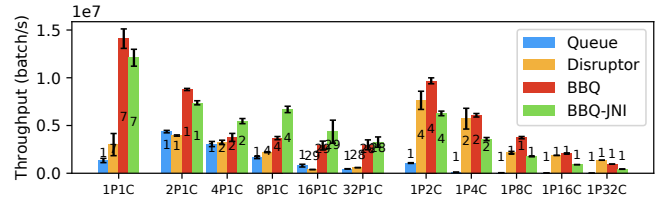


Figure 13: Throughput comparison of BBQ and BBQ-JNI against LMAX Disruptor on x86-88T.

higher throughput (14.1 Mop/s and 12.1 Mop/s, respectively). The higher performance of BBQ over BBQ-JNI is due to the JNI call overheads. With 8 producers, the difference between Disruptor and BBQ is lower (2.2 Mop/s and 3.7 Mop/s, respectively). BBQ-JNI yields 3x Disruptor's throughput (6.7 Mop/s) due to its use of `FAA`. With 32 producers, BBQ and BBQ-JNI again outperform Disruptor by an order of magnitude (3.0 Mop/s, 3.3 Mop/s, and 0.6 Mop/s, respectively).

With a single producer and multiple consumers, BBQ-JNI has no opportunity to gain performance by using `FAA`. Due to that, its performance pays the penalty of the JNI call overheads. Nevertheless, BBQ still outperforms Disruptor in most configurations. For example, BBQ yields 1.23x higher throughput than Disruptor with 2 consumers (1P2C); and 1.68x higher throughput with 8 consumers. With 32 consumers, Disruptor yields 1.42x higher throughput than BBQ.

7 Conclusion

We presented BBQ, a novel ringbuffer design that dramatically reduces the enq-deq interference by splitting the entire ringbuffer into multiple blocks. BBQ is applicable to a large spectrum of scenarios, from exchanging data to profiling, with single or multiple producers/consumers, sending fixed- or variable-sized entries, among others. Our experimental results show that BBQ outperforms several industrial ringbuffers (e.g., DPDK, LMAX Disruptor, Linux `io_uring`, Meta's Folly queue) in the great majority of workloads.

To support modern architectures such as Armv8.1, we verified and optimized BBQ with a model checker for weak memory models. Even though far from sound, verification with model checkers has proven a valuable, low-cost method of catching bugs.

Currently, our `io_uring` benchmark evaluates whether BBQ is promising for such scenarios without involving kernel details. In the future, we plan to port BBQ to kernel space to replace Linux `io_uring`.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their insightful comments. We specially thank Bohdan Trach for the helpful discussions and for proofreading this manuscript.

References

- [1] Apache Commons. <http://commons.apache.org/>.
- [2] Arm A64 Instruction Set Architecture. <https://developer.arm.com/documentation/ddi0596/2021-09>.
- [3] Arm architecture reference manual armv8, for a-profile architecture. <https://developer.arm.com/documentation/ddi0553/latest>.
- [4] Boost C++ Libraries. <https://www.boost.org/>.
- [5] BPF ring buffer. <https://www.kernel.org/doc/html/latest/bpf/ringbuf.html>.
- [6] C++ Atomic operations library. <https://en.cppreference.com/w/cpp/atomic/atomic>.
- [7] Cisco Layer Two Forwarding (Protocol) "L2F". <https://datatracker.ietf.org/doc/html/rfc2341>.
- [8] Client-Server Multi-process Example. https://doc.dpdk.org/guides/sample_app_ug/multi_process.html.
- [9] Data Plane Development Kit. <https://www.dpdk.org/>.
- [10] Data Plane Development Kit Test Suite. <https://doc.dpdk.org/dts/gsg/>.
- [11] dpdk/drivers/net/ring. <https://github.com/DPDK/dpdk/tree/main/drivers/net/ring>.
- [12] dpdk/lib/eventdev. <https://github.com/DPDK/dpdk/tree/main/lib/eventdev>.
- [13] Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.
- [14] Folly: Facebook Open-source Library. <https://github.com/facebook/folly>.
- [15] Guava: Google Core Libraries for Java. <https://github.com/google/guava>.
- [16] Intel 64 and IA-32 Architectures Software Developer's Manual. <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.
- [17] io_uring source code. https://elixir.bootlin.com/linux/v5.14-rc6/source/fs/io_uring.c.
- [18] io_uring_enter - initiate and/or complete asynchronous I/O. https://unixism.net/loti/ref-iouring/io_uring_enter.html.
- [19] Java Development Kit. <https://jdk.java.net/>.
- [20] Java Native Interface Specification. <https://docs.oracle.com/en/java/javase/11/docs/specs/jni/intro.html>.
- [21] liburing. <https://github.com/axboe/liburing>.
- [22] Linux Kernel Circular Buffers. <https://www.kernel.org/doc/html/latest/core-api/circular-buffers.html>.
- [23] LMAX Disruptor: A High Performance Inter-Thread Messaging Library. <https://github.com/LMAX-Exchange/disruptor>.
- [24] Lockless Ring Buffer Design. <https://www.kernel.org/doc/Documentation/trace/ring-buffer-design.txt>.
- [25] MPMC Queue. <https://github.com/facebook/folly/blob/main/folly/MPMCQueue.h>.
- [26] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.
- [27] Producer/consumer synchronization modes. https://doc.dpdk.org/guides/prog_guide/ring_lib.html#producer-consumer-synchronization-modes.
- [28] RISC-V. <https://riscv.org/>.
- [29] TRex: Realistic Traffic Generator. <https://trex-tgn.cisco.com/>.
- [30] Unread entries potentially lost in buf_ring after ABA condition. https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=246475.
- [31] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. Asynchronous i/o support in linux 2.5. In *Proceedings of the Linux Symposium*, pages 371–386, 2003.
- [32] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 185–192. IEEE, 2010.
- [33] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, USA, 2011.
- [34] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

- [35] David Hovemeyer, William Pugh, and Jaime Spacco. Atomic instructions in java. In *European Conference on Object-Oriented Programming*, pages 133–154. Springer, 2002.
- [36] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*, pages 96–110, New York, NY, USA, 2019. Association for Computing Machinery.
- [37] Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. Making weak memory models fair. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.
- [38] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '13*, pages 103–112, New York, NY, USA, 2013. Association for Computing Machinery.
- [39] Ruslan Nikolaev. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue . <https://github.com/rusnikola/lfqueue>.
- [40] Ruslan Nikolaev. A scalable, portable, and memory-efficient lock-free fifo queue. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- [41] Ruslan Nikolaev and Binoy Ravindran. Wcq: A fast wait-free queue with bounded memory usage. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '22*, page 461–462, New York, NY, USA, 2022. Association for Computing Machinery.
- [42] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. Vsync: Push-button verification and optimization for synchronization primitives on weak memory models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021*, pages 530–545, New York, NY, USA, 2021. Association for Computing Machinery.
- [43] Or Ostrovsky and Adam Morrison. Scaling concurrent queues by using htm to profit from failed atomic operations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 89–101, 2020.
- [44] Nicholas A Solter and Scott J Kleper. *Professional C++*. John Wiley & Sons, 2005.
- [45] Boris Teabe, Vlad Nitu, Alain Tchana, and Daniel Hagi-mont. The lock holder and the lock waiter pre-emption problems: Nip them in the bud using informed spinlocks (i-spinlock). In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 286–297, 2017.
- [46] Bill Venners. The java virtual machine. *Java and the Java virtual machine: definition, verification, validation*, 1998.
- [47] Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–13, 2016.

Sibylla: To Retry or Not To Retry on Deep Learning Job Failure

Taeyoon Kim, Suyeon Jeong, Jongseop Lee, Soobee Lee, and Myeongjae Jeon
UNIST

Abstract

GPUs are highly contended resources in shared clusters for deep learning (DL) training. However, our analysis with a real-world trace reveals that a non-negligible number of jobs running on the cluster undergo failures and are blindly retried by the job scheduler. Unfortunately, these job failures often repeat and waste GPU resources, limiting effective GPU utilization across the cluster. In this paper, we introduce Sibylla which informs whether an observed failure of DL training will repeat or not upon retry on the failure. Sibylla employs a machine learning model based on RNNs that trains on stdout and stderr logs of failed jobs and can continuously update the model on new log messages without hand-constructing labels for the new training samples. With Sibylla, the job scheduler is learning-enhanced, performing a retry for a failed job only when it is highly likely to succeed with the retry. We evaluate the effectiveness of Sibylla under a variety of scenarios using trace-driven simulations. Sibylla improves cluster utilization and reduces job completion time (JCT) by up to 15%.

1 Introduction

Deep learning (DL) has made tremendous advances in a wide range of tasks, including object detection [18], translation [32], and speech recognition [21]. To support the rapid development of DL models, enterprises typically set up a large cluster of hardware accelerators, preferably GPUs, and build a management stack to facilitate the fine-grained sharing of large-scale hardware resources. DL training often requires the use of multiple GPUs and that tasks on the allocated GPUs be scheduled simultaneously [16]. This requirement of gang-scheduling poses high communication and locality constraints on cluster management, which are less contemplated in traditional data analytics setups. In the last decade, a number of new DL cluster designs have been proposed, aiming to optimize job scheduling [5, 20, 24, 26, 34, 35], network communication [10, 16], and back-end storage systems [41], and have substantially improved cluster utilization.

In comparison, little effort has been made to address wasteful re-execution, resulting from framework support for reliable DL training. When users issue training jobs to the cluster, they wish to have their training jobs completed successfully with a high probability. The high success rate is primarily related to how effectively DL frameworks can handle job failures rooted in errors at runtime. These errors are known to occur across the stack, including infrastructure, AI engine, and user program [16, 38].

To continue training upon failure, the cluster manager can take periodic checkpoints for model weights and retry execution from the most recent checkpoint taken prior to the failure [16]. If the failure is transient and *non-deterministic* (e.g., MPI runtime failure), the job will continue training upon resuming from the checkpoint, as transient issues are not supposed to repeat [16]. However, this approach does not help recover from *deterministic* failures (e.g., syntax or configuration errors), as the same faulty condition will recur while re-running the failed job. DL training that experiences these two types of failures implies that retrying job executions on deterministic failures would waste GPU cycles. Our characterization study shows that the resource inefficiency caused by the unnecessary retries is non-negligible (§ 2).

Our approach. We introduce a case for *learning-enhanced* job scheduling that substantially reduces unnecessary job retries. Our system, Sibylla, *predicts* whether a failed job deserves a retry or not. Since failures can occur anywhere across the stack, the prerequisite for failure prediction with high accuracy is to collect a training dataset that faithfully reflects failure-related information. We use standard error streams of training jobs directed into log files (i.e., stdout and stderr) – every software stack makes use of these error streams to record execution path/status- and error-related information. Sibylla employs a recurrent neural network (RNN)-based DL model to train the log files and build a failure classifier. Sibylla further automates the process by adopting auto-labeling that allows new training samples to be incorporated without hand-constructing their labels. With this technique, Sibylla can automatically update the model upon aggregating new log

messages from recently completed jobs without access to human experts for labeling data (§ 3).

One major contribution of Sibylla is that it avoids using a static way for failure classification, which would be impractical in the long run. For example, as log formats are unstructured and diverse, grepping log files for specific keywords that are endemic to deterministic/non-deterministic failures, as done in [16], requires someone to keep identifying new error-related expressions, which is too manual and time-consuming. We even tried a clustering model as a non-DL approach to automatically group failures and classify their types based on similar words, but the accuracy is much lower than our DL-based approach (§ 4).

Results. We evaluate Sibylla through simulations [10] using traces derived from a Microsoft production cluster [2]. We account for the effectiveness in both the best case and the worst case, where avoiding retries on deterministic failures guided by our predictor is associated with the longest-running jobs and the shortest-running jobs, respectively. Sibylla reduces the average job completion time (JCT) by 6.5–15.4% for the best case and 3.6–10.5% for the worst case.

2 Job Failures in Deep Learning Cluster

DL platform overview. A shared cluster for deep learning training typically consists of a number of multi-GPU machines that constitute a pool of hundreds to thousands of GPUs. The GPU machines are connected to a high-speed network (e.g., 100-Gbps InfiniBand) to speed up distributed training that requires multiple GPUs. The cluster scheduler has an objective to decide the jobs to run next (e.g., minimizing JCT) and a strategy for placing the jobs on available GPUs (e.g., preferably onto the same machine). Docker container is used to isolate CPU, GPU, and memory resources between concurrent jobs. For distributed training, we are based off data parallelism that performs model synchronization via either parameter servers or collective communication libraries (e.g., MPI [7], NCCL [25]). A back-end distributed storage system is dedicated to storing stdout and stderr logs generated during training across the entire cluster. Target DL applications are run on popular engines like TensorFlow [3] and PyTorch [27].

Deterministic vs non-deterministic. Failures come from the job scheduler, the storage system, and other components that constitute the DL platform. We categorize failure occurrences into either deterministic (DT) or non-deterministic (NDT) to determine retry on failure. Table 1 shows how existing DL job failures [16, 38] can be classified into these two categories according to failure reasons.

Deterministic failures (or *DT failures*) are caused by inherent code syntax errors, API misuse, misconfigured settings, etc. For example, a job may try to load non-existent data, data in an inconsistent format, or data in corruption. Alternatively, a job may use a library version that the platform does not

Type	Category	Failure Reason Examples
DT	Deep Learning Specific	Framework API Misuse, Tensor Mismatch
	Environment Error	Path/Library Not Found, Permission Denied
	Code Error	Key/Attribute Not Found, Illegal Arguments
	Data Error	Corrupted Data, Unsupported Encoding
NDT	CPU OOM	CPU Out of Memory
	GPU OOM	GPU Out of Memory
	Runtime Error	MPI Daemon Failure, Network Conn. Failure
	Node Error	Unexpected Worker Node Exited

Table 1: Failure classification and failure reason examples.

support or has dependency issues. Jobs experiencing these failures will end up in unsuccessful training as the failures repeat.

On the contrary, *non-deterministic failures* (or *NDT failures*) are accidental and usually related to temporal network connection loss or transient issues of the job’s assigned node. For example, workers of a distributed training job may not communicate with each other due to network outages or MPI daemon errors on the host machine. Or, a job may use host memory more than allowed and want to be scheduled on a larger machine. Retry from failure helps overcome this type of failure.

Failure handling today. Due to the intricate process of failure classification, job schedulers today are utterly ignorant of the type of failure that occurred and takes simple heuristics for failure handling. Failed jobs in Microsoft Philly [16] are retried a fixed number of times to overcome NDT failures and successfully complete more jobs after retries. To facilitate this, in Philly each job is configured to create a model checkpoint after finishing a certain number of epochs. On the other hand, NoRetry¹ in a large enterprise terminates every job that experiences a failure to avoid worthless re-execution of jobs in DT failure.

However, these approaches face significant challenges that limit their merits: (1) Philly cannot prevent GPU cycles wasted by DT failures; (2) NoRetry cannot achieve as good training productivity as Philly because it terminates all NDT failure jobs that deserve retries for successful training. In addition, the retry mechanism in NoRetry greatly obfuscates our understanding of the reasons behind failures between DT and NDT, affecting user experience.

A DT failure repeats regardless of how the scheduler places the job on GPUs, whereas an NDT failure may not repeat after a new scheduling attempt. Therefore, we also call them repetitive failure versus non-repetitive failure. Although some jobs terminate quickly during DT failures, there are DT failures that take a fairly long time for the failures to be manifested (e.g., incorrect data inputs).

Opportunities. In this paper, we propose a failure classifier using machine learning to separate deterministic and non-deterministic failures at runtime. To reveal opportunities for using it for predictive retry, we conduct workload character-

¹Anonymized upon request by the company.

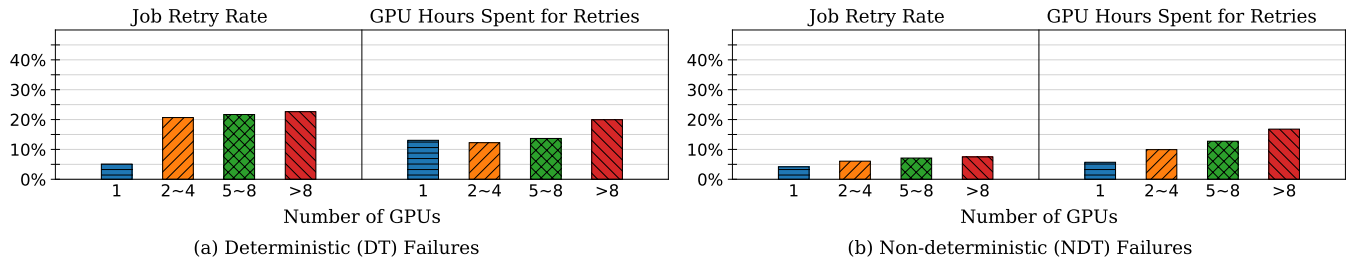


Figure 1: Job retry rates and the fraction of GPU hours spent during retries for DT and NDT over different job sizes.

ization using an openly available Philly trace [2]. The trace contains information about each training job, including each attempt of job scheduling, GPUs allocated for the attempt, the start and end time of job execution during the attempt, and the job’s final completion status. As a scheduling attempt occurs in both the initial job issue and subsequent retries on failures, using the trace we can estimate job retry rates (i.e., # jobs experiencing retry ÷ # all jobs) and the fraction of GPU hours spent during retries out of all GPU hours. Figure 1 shows the results for two failure types, DT and NDT, over different job sizes based on the GPU request distribution.

First, in Figure 1(a), we observe that jobs are frequently exposed to DT failures and thus waste a significant portion of GPU hours due to useless retries. Specifically, jobs that use more GPUs retry execution more often while GPU hours consumed during retries account for 12.3–19.9% across job sizes. This is the amount of GPU hours wasted by Philly, yet can be saved by an optimal predictive retry. Moreover, as previously stated, DT failures could exhibit high run times to failure (RTF). In particular, for failed executions, the median RTF is 614 and 2,458 seconds for DT and NDT, respectively, with the 80th-percentile increasing to 6,037 seconds for DT and 34,133 seconds for NDT.

NoRetry does not waste these GPU cycles at all since no retry occurs. However, Figure 1(b) implies that the training success rate in NoRetry will go down by around 4.5% since all NDT failure jobs are doomed to be aborted. To circumvent NDT failure, users will need to resubmit those jobs to the cluster and restart training from the initial state. Such restarting indicates that the GPU hours spent in the previous job executions before the failures become wasteful.

Based on the observations, we believe cluster utilization and reliability of DL platforms can be enhanced by performing job retry only when predicted as non-repetitive, guided by a failure classifier. The idea of adapting job retry based on failure type is not new but instead has been presented merely as a design implication [16, 28, 38]. To the best of our knowledge, our work is the first to evaluate its feasibility.

3 Sibylla Design

Sibylla is an RNN-based prediction system that has the following design goals.

- **High accuracy.** Sibylla should achieve high prediction accuracy for both types of failures. Otherwise, mispredictions can lead to low cluster efficiency or low training success rates.
- **Ease of use.** It is cumbersome to build a new training dataset every time new failure samples are generated. Once a prediction model is built, Sibylla provides an option to label new failure samples and re-train the model automatically.
- **Ease of integration.** Sibylla operates in a stand-alone agent or runs on the application side (e.g., Application Master in Apache YARN) to interact with the scheduler. The scheduler only needs to send a prediction input to Sibylla and get notified with the output (DT or NDT). Sibylla does not interfere with the scheduler’s main tasks, such as job placement.

Samples for training. We use stdout and stderr log messages to train Sibylla. These logs record the execution information of the software stack and have been widely used in anomaly detection and distributed system or software troubleshooting scenarios [13, 19, 29, 36]. Similarly, every software stack in the DL cluster records execution- and error-related information in standard error streams. So, stdout and stderr logs are our choice of training data in Sibylla. However, using messages in the log poses a critical challenge: log messages are unstructured and contain many redundant and uninteresting lines of text to exclude.

Training workflow. Figure 2 illustrates the workflow of Sibylla. It first performs data preprocessing to extract useful log sequences and convert them into semantic vectors. Then, our RNN-based models are trained with these vectorized inputs. Sibylla includes an additional auto-labeling stage based on a reliable ensemble method to learn new incoming data.

Step 1) Data preprocessing. Because the log file size is typically non-uniform, it is necessary to transform each original log to be uniformly sized. A log often indicate failure symptoms at the line with relevant keywords (such as failure, error, etc). With this insight, Sibylla takes up to 5 lines after the line where such a keyword is present. Sibylla also includes some lines preceding the keyword as they may indicate a log sub-sequence that leads to failure. We empirically tested a variety of line lengths and landed on 20 lines because this is overall the minimum number of lines producing the highest prediction accuracies. On not observing the failure keyword,

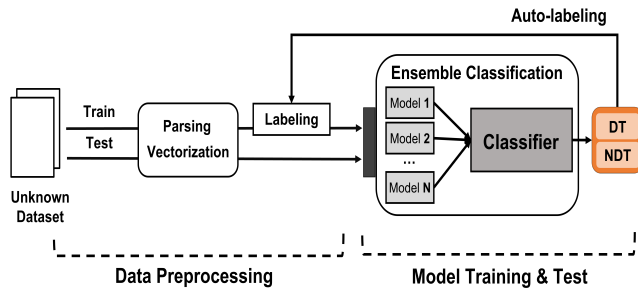


Figure 2: Overall architecture of Sibylla.

Sibylla takes the last 20 lines from the log file as an input for further preprocessing.

Since the log is in an unstructured plain-text form, we need *parsing* and *vectorization* stages to extract semantic information, as shown in Figure 3. At the parsing stage, each log is categorized into a structured *template* that primarily removes words unrelated to the semantics, such as non-character words and stop words. The structured template is thus informative enough to represent the original text. Sibylla applies the state-of-the-art parsing tool called Drain [12], which has been widely exploited in prior log-based analysis studies for its superior effectiveness [11, 22, 37, 39]. A structured text template is transformed into a *semantic vector* and fed into the training model. This vectorization process first digitizes each word into a vector. It then accumulates all word vectors of each line in the template into a single semantic vector entry by weighing each word based on TF-IDF (term frequency-inverse document frequency) score. Sibylla uses the FastText algorithm [17] to extract a semantic information across the log.

Step 2) Model training. The semantic vector sequences serve as an input to model training. There are two representative RNN models involved in training Sibylla: bi-directional long short-term memory (LSTM) and attention-based gated recurrent unit (GRU).

Training a log-based detection model can be supervised, unsupervised, or semi-supervised. Supervised learning ensures that the model achieves high performance, but this approach necessitates all data to be labeled ahead of time. However, cluster job executions generate a significant amount of log data, making it infeasible to have domain experts label all DT and NDT failure samples for supervised learning. Instead, unsupervised learning can proceed with fully unlabeled data but usually sacrifices model performance. Sibylla adopts semi-supervised learning. It starts model training with partially labeled data and keeps updating the model with unlabeled data by auto-labeling them in an online fashion.

Automatic sample labeling. For auto-labeling to be effective, the classifier is required to make a robust decision. In other words, the classifier should make good decisions even for

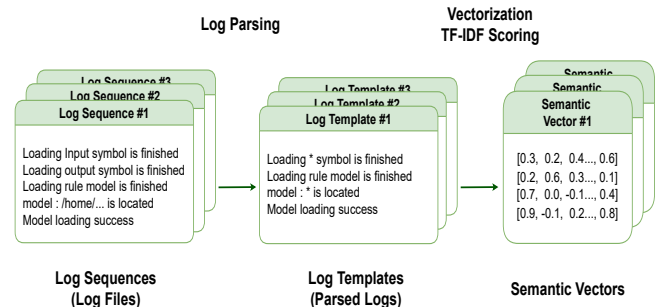


Figure 3: Data preprocessing steps in Sibylla.

unseen data that may confuse the classifier. Sibylla automates the labeling process by allowing the classifier to leverage the prediction results of multiple RNN models with different architectures. It takes advantage of an ensemble method that performs voting on the prediction results to decide the failure type, mitigating the effect of a single wrong prediction. Specifically, Sibylla trains K RNN models independently ($K = 2$ in our default setup) and makes a classification decision by aggregating information from individual models regarding the predicted failure type. The final decision is made through a majority voting mechanism, where each model has an equal weight of reflecting its decision on DT versus NDT.

Integrating into cluster managers. There are two tasks to be done by cluster managers to use Sibylla. First, when a failure of a DL job occurs with a stdout/stderr log containing an error, the cluster manager transmits it to Sibylla and receives the notification of the expected failure type. Second, the cluster manager delivers a batch of log files with labels to build an initial model or files without labels to improve the model on observing new failures. Note that our main focus in this paper is on presenting the design principle of predictive retry. Nonetheless, we believe Sibylla can be imported into commodity GPU cluster managers without significant hurdles.

4 Evaluation

We present Sibylla’s accuracy (§ 4.1) and JCT improvements using a GPU cluster simulator with Philly trace (§ 4.2).

Dataset. Since no dataset for failed DL jobs is publicly released, we construct one that contains most of the known failures. We obtained 97 failure log files from the company operating NoRetry and collected additional 159 failure messages through a manual search on Stack Overflow [1], including 20 out of 21 failure categories (w/o GPU ECC error) presented in [16]. We then apply data augmentation to enlarge a training dataset while retaining key properties of the data. For our scenario, two popular text augmentation methods, WordNet [23] and Word2Vec [9], are used to replace words in an original log file with cognitive synonyms and create a new augmented

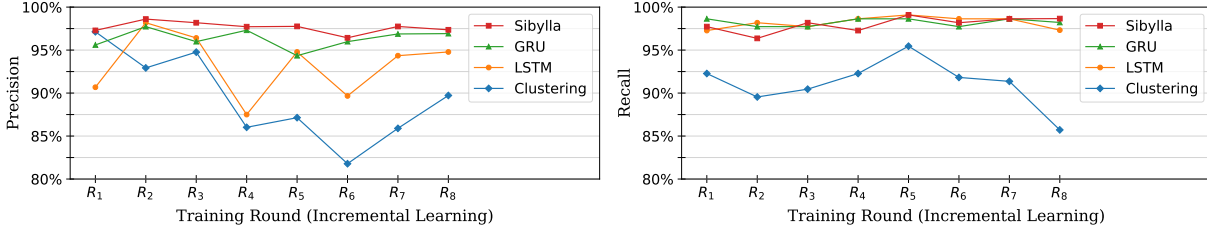


Figure 4: Precision and recall for NDT over training rounds.

file. As a result, we have 4468 log files as a dataset.

Although NDT failures are fewer in number than DT failures in reality, our dataset is augmented such that samples are balanced between DT and NDT and across failure categories [38]. This sample balancing is mainly to make the decision boundary of the model not biased [4, 6, 30, 31, 33]. Further, DL applications appearing in the data are diverse, e.g., image classification, language model, and audio recognition, and run on popular engines like TensorFlow and PyTorch.

Accuracy metrics. For DT/NDT, classifier accuracy is measured using *precision* (fraction of predictions that are truly deterministic/non-deterministic failures) and *recall* (fraction of true deterministic/non-deterministic failures predicted correctly by the classifier). Thus, for both precision and recall, higher is better.

4.1 Classifier Performance

Accuracy		Clustering	RNN Model		Sibylla	Oracle
			LSTM	GRU		
NDT	Precision	89.72	94.78	96.92	97.36	98.66
	Recall	85.71	97.32	98.21	98.66	98.66
DT	Precision	86.67	97.32	98.24	98.68	98.70
	Recall	90.43	94.78	96.96	97.39	98.70

Table 2: Final accuracy among competing classifier designs.

Experiment process. We assess the effectiveness of Sibylla over training on multiple insertions of new log data. For this evaluation, we split the dataset into ten partitions $\{p_1, p_2, \dots, p_{10}\}$ and go through eight rounds of training $\{R_1, R_2, \dots, R_8\}$. Each round has training, validation, and test data, where training and validation data are used for model training, while test data is used to report prediction accuracy (i.e., precision and recall). As the round moves on, Sibylla auto-labels the previous test data and uses it as new validation data. To illustrate, in the first round (R_1), we use p_1 , p_2 , and p_3 as training, validation, and test data, respectively. With proceeding to R_2 , the next unused partition (p_4) becomes new test data, while training and validation data are reorganized into $p_1 + p_2$ and p_3 , respectively. Here, p_3 is auto-labeled by Sibylla. Continuing this process will report accuracy incrementally over eight rounds of training.

Results. We compare Sibylla using the proposed ensemble classifier with other classifier designs based on a single NN model (LSTM and GRU) and a non-NN model (Clustering). Currently, our ensemble model is created by combining two models, LSTM and GRU. Since these classifiers serve as auto-labeling, as a baseline we include an oracle method in which all data used for model training are labeled 100% correctly. Table 2 shows the prediction accuracy of all competing classifiers observed in the final round of the incremental training (i.e., R_8) for both DT and NDT. The results show that as compared to the clustering method, NN-based classifiers achieve overall higher accuracy for predicting both DT and NDT. Among NN-based classifiers, Sibylla obtains the highest accuracy while approaching the closest to the oracle’s performance.

In Figure 4, we show how the prediction accuracy for NDT changes during the incremental training – DT has similar trends. Our ensemble approach provides prediction with consistently higher precision over training rounds, with Clustering significantly worse than others as expected. For recall, there is no substantial difference among NN-based methods. As higher precision and recall are always desirable, we prefer an ensemble approach over approaches using a single model for classification and auto-labeling.

4.2 Simulation Results

Setup. Next, we evaluate our predictive retry while replaying the Philly trace on the GPU cluster simulator designed for prior work [10]. We use three job scheduling policies, smallest-job-first w.r.t. GPU requirement (SJF), 2D-LAS (DLAS), and 2D-Gittins index (GITTINS) [10], to schedule jobs in the trace. The cluster comprises 200 nodes, each having 8 GPUs, 256 GB of host memory, and 64 CPU cores.

As the trace mainly contains information about each job’s retry and final status without its log messages, we choose to apply our classifier created from our dataset (with the prediction accuracy in Table 2) considering the worst (**Worst**), average (**Average**), and best case (**Best**). For Worst and Best, we apply misprediction to the longest-running and shortest-running jobs, respectively – so, the penalty of misprediction is the highest versus the lowest. For Average, we select the jobs experiencing misprediction randomly. We have two baselines

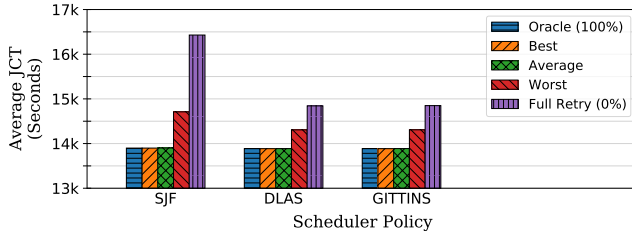


Figure 5: JCT reductions for using different schedulers.

to compare with Sibylla: **Oracle**, which makes 100% correct predictions, and **Full-Retry**, which retries jobs without prediction as done in Philly. For a fair comparison, we take into account the average JCT (including the queuing time) of successfully completed jobs only.

As Figure 5 shows, all adaptive retry strategies can reduce the average JCT compared to the conventional method, Full-Retry. Best improves the JCT by 15.4% for SJF, 6.5% for DLAS, and 6.5% for GITTINS, and even Worst reduces the JCT by 3.6–10.5%. Moreover, strategies based on Sibylla (i.e., Best, Average, and Worst) are, on average, only 1.3% worse than Oracle, which delivers the most JCT reduction. With high accuracies, Sibylla has a negligible impact on the job success rate resulting from mispredicting NDT as DT. Our recall for predicting NDT is 98.66%, lowering the job success rate by only 0.06% from 75.04%.

5 Related Work

Machine learning-based anomaly detection. Prior work studies using machine learning on textual log data obtained from various systems such as HDFS (Hadoop Distributed File System) and BGL (BlueGene/L) [14] to detect abnormal and anomalous system behaviors. DeepLog [8] adopts an LSTM model [15] for anomaly detection and diagnosis. It first trains the model on normal log messages to learn their sequences and uses the model to recognize abnormal sequences from online log data for anomaly detection. LogAnomaly [22] concatenates log sequences as a template to extract more precise log semantics and applies the anomaly detection method similar to DeepLog. LogRobust [39] leverages existing learned word collections such as Word2Vec [9] to analyze various unstructured logs and measures abnormal logs using Attention-based Bidirectional LSTM [40]. PLELog [37] proposes semi-supervised learning based on probabilistic label estimation to make the sample labeling process more practical.

These strategies aim to detect abnormalities in large-scale system logs through deep learning. Despite similarities, our work is differentiated in that it focuses on predicting repetitive DL job failures and assessing how such capability helps improve job completion times in shared GPU clusters.

DL job failure analysis. Our work is motivated by numer-

ous works that reveal the cluster inefficiency caused by DL job failures [16, 38]. These works analyze logs for program failures of industrial jobs from Microsoft Philly, whose public scheduler log is used for our study. They investigate the categories and root causes of job failures, suggesting that current practices of failure handling in DL platforms can be enhanced. Although they are the first to stress the necessity of an adaptive retry mechanism driven by the failure type, no prior work has faithfully evaluated its feasibility.

6 Concluding Remarks

To deal with DL job failures, it is critical to precisely predict whether the current failure will repeat or not upon re-execution. Our RNN-based predictor, Sibylla, correctly informs this repetition potential, enabling the cluster scheduler to incorporate it to perform an adaptive retry on failure. With Sibylla, today’s DL platforms not only reduce resource waste by avoiding retries for repetitive failures but also retain job productivity by continuing job executions for transient failures. We confirm this efficacy with trace-driven simulations.

Future works. Misprediction for deterministic failure may repeat when a job produces similar messages over retries. To avoid this repetitive misprediction, we could revise the classifier to incorporate the feedback about predictive re-execution from the cluster scheduler.

Another interesting future work is extending our method for new failure types that have not occurred. An assumption we made in the design of Sibylla is that a new type of failure with unseen semantics does not appear. However, failure message formats from online logs could be diverse as developers’ message logging practice is personalized and unstructured. We have done a brief study on how destructive unknown failure types are by measuring accuracy when a new failure type appears in the middle (7th round) of training from Figure 4. It turns out that the classification accuracy of Sibylla for deterministic failure can drop to 40%, especially for precision. Thus, we may need to incorporate human experts for labeling new data in low prediction confidence rather than relying on auto-labeling. Nonetheless, we think the chance of observing new types of failures is somewhat low.

Acknowledgements

We thank our anonymous shepherd and reviewers for their valuable comments and suggestions. We also thank Xinyue Ma from UNIST for helpful feedback and comments. This work was supported by Samsung Data & Information Technology Center, Kakao Brain Corporation, Rebellions Inc., and the U-K BRAND Research Fund (1.220028.01 and 1.220036.01) of UNIST(Ulsan National Institute of Science & Technology).

References

- [1] Stack Overflow, 2008. <https://stackoverflow.com/>.
- [2] Msr-fiddle/philly-traces, 2019. <https://github.com/msr-fiddle/philly-traces>.
- [3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.
- [4] Mateusz Buda, Atsuto Maki, and Maciej A. Mazurowski. A Systematic Study of the Class Imbalance Problem in Convolutional Neural Networks. *Neural Networks*, 2018.
- [5] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning. In *EuroSys*, 2020.
- [6] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of artificial intelligence research*, 2002.
- [7] Leonardo Dagum and Ramesh Menon. OpenMP: an Industry Standard API for Shared-Memory Programming. *IEEE computational science and engineering*, 1998.
- [8] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly Detection and Diagnosis from System Logs Through Deep Learning. In *CCS*, 2017.
- [9] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.
- [10] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*, 2019.
- [11] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. An Evaluation Study on Log Parsing and Its Use in Log Mining. In *DSN*. IEEE, 2016.
- [12] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An Online Log Parsing Approach With Fixed Depth Tree. In *ICWS*. IEEE, 2017.
- [13] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Experience Report: System Log Analysis for Anomaly Detection. In *ISSRE*. IEEE, 2016.
- [14] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Loghub: A Large Collection of System Log Datasets Towards Automated Log Analytics. *arXiv preprint arXiv:2008.06448*, 2020.
- [15] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-term Memory. *Neural computation*, 1997.
- [16] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *ATC*, 2019.
- [17] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Herve Jégou, and Tomas Mikolov. FastText.zip: Compressing Text Classification Models. *arXiv preprint arXiv:1612.03651*, 2016.
- [18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012.
- [19] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining Invariants from Console Logs for System Problem Detection. In *ATC*, 2010.
- [20] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and Efficient GPU Cluster Scheduling. In *NSDI*, 2020.
- [21] Avner May, Alireza Bagheri Garakani, Zhiyun Lu, Dong Guo, Kuan Liu, Aurélien Bellet, Linxi Fan, Michael Collins, Daniel Hsu, Brian Kingsbury, et al. Kernel Approximation Methods for Speech Recognition. *arXiv preprint arXiv:1701.03577*, 2017.
- [22] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs. In *IJCAI*, 2019.
- [23] George A Miller. WordNet: A Lexical Database for English. *Communications of the ACM*, 1995.
- [24] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *OSDI*, 2020.
- [25] NVIDIA Collective Communications Library (NCCL), 2017. <https://docs.nvidia.com/deeplearning/nccl/>.
- [26] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *EuroSys*, 2018.
- [27] PyTorch, 2018. <https://pytorch.org/>.
- [28] Junjie Qian, Taeyoon Kim, and Myeongjae Jeon. Reliability of Large Scale GPU Clusters for Deep Learning Workloads. In *WWW*, 2021.
- [29] Barbara Russo, Giancarlo Succi, and Witold Pedrycz. Mining System Logs to Learn Error Predictors: A Case Study of A Telemetry System. *Empirical Software Engineering*, 2015.
- [30] Connor Shorten and Taghi M Khoshgoftaar. A Survey on Image Data Augmentation for Deep Learning. *Journal of big data*, 2019.
- [31] Connor Shorten, Taghi M Khoshgoftaar, and Borko Furht. Text Data Augmentation for Deep Learning. *Journal of big Data*, 2021.
- [32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *NIPS*, 2017.
- [33] Jason Wei and Kai Zou. Eda: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks. *arXiv preprint arXiv:1901.11196*, 2019.
- [34] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, Lidong Zhou, Gandiva: Intropective Cluster Scheduling for Deep Learning. In *OSDI*, 2018.
- [35] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *OSDI*, 2020.
- [36] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting Large-scale System Problems by Mining Console Logs. In *SIGOPS*, 2009.
- [37] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. Semi-supervised Log-based Anomaly Detection via Probabilistic Label Estimation. In *ICSE*, 2021.
- [38] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. An Empirical Study on Program Failures of Deep Learning Jobs. In *ICSE*. IEEE, 2020.
- [39] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. Robust Log-based Anomaly Detection on Unstable Log Data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.
- [40] Peng Zhou, Wei Shi, Jun Tian, Zhenyu Qi, Bingchen Li, Hongwei Hao, and Bo Xu. Attention-based Bidirectional Long Short-term Memory Networks for Relation Classification. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 2: Short papers)*, 2016.

- [41] Yue Zhu, Weikuan Yu, Bing Jiao, Kathryn Mohror, Adam Moody, and Fahim Chowdhury. Efficient User-level Storage Disaggregation for Deep Learning. In *CLUSTER*, 2019.



Speculative Recovery: Cheap, Highly Available Fault Tolerance with Disaggregated Storage

Nanqinqin Li, Anja Kalaba, Michael J. Freedman, Wyatt Lloyd, and Amit Levy
Princeton University

Abstract

The ubiquity of disaggregated storage in cloud computing has led to a nascent technique for fault tolerance: instead of utilizing application-level replication, newly-launched backup instances recover application state from disaggregated storage (REDS) after a primary's failure. Attractively, REDS provides fault tolerance at a much lower cost than traditional replication schemes, wherein at least two instances are running. Failover in REDS is slow, however, because it sequentially first detects primary failure and only then starts recovery on a backup.

We propose *speculative recovery* to accelerate failover and thus increase the availability of applications using REDS. Instead of proceeding with failover sequentially, speculative recovery safely and efficiently parallelizes detecting primary failure and running recovery on a backup, by employing our new `super` and `collapse` primitives for disaggregated storage. Our implementation and evaluation of speculative recovery demonstrate that it considerably reduces failover time.

1 Introduction

Replicated, network-attached storage devices have all but replaced local disks in cloud settings. Such *disaggregated disks* provide a host of useful features, including scalable storage capacity and performance, convenient data backup, and disk fault-resilience [16, 33, 49].

Their ubiquity has led developers to begin leveraging *disk* fault-tolerance to achieve *application* fault-tolerance [12, 15, 50]. When an application running on one instance, the primary, uses a disaggregated disk, its state survives its failure. This enables an emerging fault-tolerance technique we term *recovery from disaggregated storage* (REDS) where a backup instance recovers application state from the disaggregated disk and continues serving the application. In general, single-node applications can use REDS unmodified as long as they are crash-consistent—i.e. they persist state updates to disk before *externalizing* them to clients and are able to recover state from disk after a crash [27, 39, 58]. This includes most relational databases, local key-value stores, and file systems.

REDS is an alternative to the traditional *application-level replication*, where the application running on the primary con-

tinuously replicates its state to at least one backup [20, 22, 24, 42, 44, 56, 62, 66]. Application-level replication provides *high availability* since it ensures that a backup can service requests immediately should the primary fail. However, application-level replication is also expensive as each backup runs an entire instance of the application, requiring as much CPU, memory, storage, network resources, etc., as the primary.

In contrast, REDS only requires running a single instance of the application at a time but sacrifices availability since failover can be slow. In particular, REDS requires that the disaggregated disk be detached from a potentially faulty instance *before* initiating recovery on a new one. As a result, REDS risks long recovery periods on the new instance when the original may have come back online faster, e.g., when a transient networking issue resolves itself, or waiting too long to determine the original instance has indeed failed.

In this paper, we introduce *speculative recovery*, an application fault-tolerance technique that leverages disaggregated disks to achieve resource efficiency similar to REDS with significantly higher availability. Speculative recovery begins as soon as the primary *appears* unavailable, e.g., when it stops responding to health checks. It immediately begins recovery on a new backup instance by creating an independent clone of the disk and attaching it to the backup, while the primary instance is not interfered with to allow it an opportunity to come back in parallel. Whichever instance, the primary or the backup, becomes available first serves the application while the other is deallocated. This reduces unavailability to the minimum of either the primary becoming available again or the backup's speculative recovery completing.

There are two major challenges in realizing speculative recovery on existing disaggregated storage systems. The first is ensuring application correctness, i.e., linearizability [37], when both the primary and the backup are using a clone of the same application disk. This requires that updates to the disk from one instance do not interfere with the other, and that the external world only ever sees the effects of updates from one instance. The second challenge is ensuring good disk performance for the backup instance to recover the application. Many existing disaggregated storage systems have designs for disk clones that provide poor performance.

To address these challenges, speculative recovery introduces new primitives, `super` and `collapse`, for disaggregated disks. `super` allows a disk to be in a *superposition* temporarily where two independent versions of the disk are allowed to diverge until a `collapse` when one is observed, and it appears as though the other never existed. In essence, `super` provides disk clones with isolation and good performance, and `collapse` guarantees correctness by ensuring only one, primary or backup, of the clones can be observed.

`super` uses copy-on-write to achieve effective isolation, and the ephemeral nature of superposition enables a new design we term *collocated-clone* that minimizes the negative performance impact of copy-on-write. With *collocated-clone*, a disk clone directly refers to its parent's allocation table to locate data blocks, eliminating the overhead of re-populating the clone's own allocation table, which is a major bottleneck in some existing disaggregated storage systems. *Collocated-clone* also adopts a minimal data path by keeping all data blocks of a clone on the same storage shards as the corresponding blocks of its parent. We believe that such collocation does not skew the data distribution of a storage cluster given that only one clone continues after `collapse`.

`collapse` uses a *dirty bit* to ensure only one clone of the disk is ever externally observable. The dirty bit reflects whether there have been any updates to the disk from the primary after `super` is invoked. If so, `collapse` determines that the primary may have been observed and then aborts speculative recovery by deallocating the disk clone and the backup. Otherwise, `collapse` ensures no future writes from the primary will be accepted and then informs the backup that it can start externalizing state updates.

We implement `super` and `collapse` based on Ceph [68], an open-source distributed storage system, and use them to implement speculative recovery from disaggregated storage (SpecREDS). Our evaluation compares SpecREDS to REDS for three stateful applications: MySQL, PostgreSQL, and MariaDB. We find that our *collocated-clone* design achieves near-normal disk performance that supports application recovery up to an order of magnitude faster compared to Ceph's native clone design. Such improvement enables SpecREDS to achieve significantly faster failover in some scenarios.

In summary, the main contributions of this paper include:

- Speculative recovery, which increases the availability of applications that achieve cheap fault tolerance using REDS.
- The `super` and `collapse` primitives and their designs including *collocated-clone* for disk cloning with near-normal performance and the dirty bit for guaranteeing correctness.

2 Highly Available Applications

Stateful data center applications strive to provide high availability in the face of individual machine failures. This is often

exacerbated on the cloud because developers may have no way to recover data from a virtual machine's or a container's disk after a failure. Practitioners today adopt both traditional fault-tolerance techniques at the application-level as well as cloud-native techniques that rely on disaggregated storage.

2.1 Application-level Replication

A standard approach to highly available fault tolerance for stateful applications is to replicate the application across multiple compute instances (physical machines, VMs, containers, etc.). Commonly, applications use primary-backup replication where a primary instance handles all client requests and forwards the execution logs to backup instances. If the primary fails, backups are ready to be promoted with minimal overhead since their local state is already up-to-date.

However, application-level replication has two major drawbacks. First, it can be costly. Because backups require redundant compute resources—CPU, memory, etc.—adding a backup costs as much as hosting the original application. Second, support for application-level replication is often implemented separately for each application [53, 56]. While many stateful applications support replication, including MySQL, PostgreSQL, and MongoDB, many do not, including SQLite, LevelDB, and RocksDB.

2.2 Recovery From Disaggregated Storage

Two recent trends have enabled alternative fault-tolerance strategies. First, cloud platforms have adopted disaggregated storage [28, 41, 45, 52] to provide virtual block devices to enable more efficient resource management and provide more reliable services [16, 33, 49]. Data stored on these disaggregated disks is striped and replicated across a storage area network to provide highly available and highly durable block devices that can outlast failures of the compute instances they are attached to. Second, provisioning compute instances (VMs or containers) has become fast—new compute instances can be spawned in seconds rather than in minutes [2, 5, 13, 47].

As a result, practitioners have adopted an alternative fault-tolerance mechanism, REDS, leveraging disaggregated disks and fast provisioning [12, 15, 50]. In REDS (Figure 1), instead of maintaining live backup replicas of the application, a backup instance is only spawned after the primary instance is presumed down. The disaggregated disk is then moved from the failing primary to the new backup and the application is restarted on the backup. Since the application data stored on disk persist through machine failures, the backup can recover the application to a consistent pre-failure point.

REDS provides fault tolerance to stateful applications at virtually no additional cost, since only a single instance is provisioned most of the time, with at most a short overlap of a primary and backup instance during failures. Moreover,

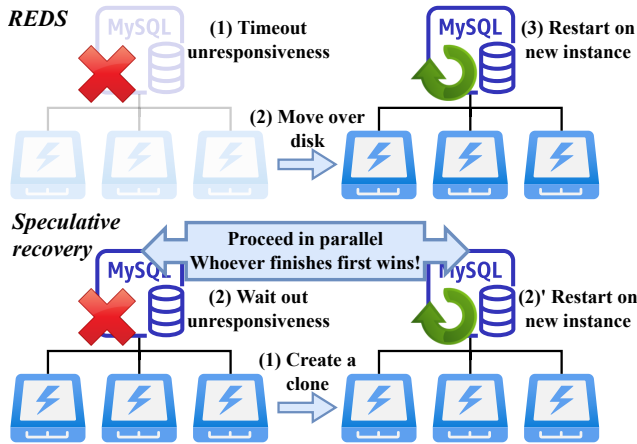


Figure 1: **REDS vs speculative recovery.** REDS sequentially times out the unresponsiveness and restarts the application on a new instance by moving over the application disk, whereas speculative recovery parallelizes the two instances.

unlike application-level replication, it does not require explicit support from the application and can thus support any crash-consistent [27, 39, 58] application—i.e., any application that persists state changes before externalizing them and can recover to a consistent state from disk following a crash failure. This includes most relational databases, local key-value stores, and file systems.

2.2.1 Lower Application Availability

Compared to application-level replication, REDS suffers from lower availability due to the relatively long process of restarting the application after failure has occurred.

As shown in Figure 1, the failover process in REDS includes two steps: (1) determining whether the primary instance has failed and (2) recovering the application on a backup instance. Step 1, the timeout phase, is typically achieved using a timeout of unresponsiveness for the primary to avoid spurious downtime during Step 2. Step 2 recovers the application by spawning a new instance as the backup, moving over the application disk, and restarting the application.

These steps *must* happen sequentially since both require exclusive access to the disk. For the timeout phase, the primary needs the disk attached in case it becomes responsive again. For the recovery phase, the backup needs the disk to restart the application. As a result, downtime following a failure is dictated by the sum of the timeout and the recovery phase.

Clearly, short recovery and short timeouts would improve availability. Much of recovery—spawning a new virtual machine or container and attaching a disaggregated disk to these instances—is relatively fast and becoming faster in modern data center infrastructure. For example, an AWS EC2 virtual machine can be allocated and spawned in a few seconds with optimized operating system distributions [57], while

containers as well as other cloud virtualization techniques can allocate runtime environments an order of magnitude faster [13, 47]. Similarly, disaggregated disks, such as a Ceph block device, can be attached to a new virtual machine or container in a few hundred milliseconds.

Application recovery time, on the other hand, is less predictable—the same application might require a few seconds or several minutes to recover depending on, e.g., the state of the application’s write-ahead-log. As a result, very short timeouts risk triggering such long recoveries unnecessarily when the primary’s unresponsiveness is ephemeral (e.g. the monitor is faulty, a packet is lost, etc.). For instance, if a temporary network problem leads to 6 seconds of unavailability for the primary, using a short 5 second timeout to trigger a 1 minute recovery leads to 65 seconds of unavailability. In contrast, a longer timeout would have only 6 seconds of unavailability in this scenario. In practice, “primary-is-failed” timeouts are often quite long—for example, Kubernetes uses a default 5 minute timeout to detect node failure [43].

3 Introducing Speculative Recovery

REDS can result in poor availability because the primary must be marked irreversibly failed before recovery can be attempted on a backup. Timeout lengths are chosen conservatively to avoid long recoveries if the failure is temporary, and recovery cannot begin until *after* this timeout is expired. In the worst case, this results in downtime during a long timeout followed by more downtime until a long recovery completes.

This is fundamental to REDS because there is no way to predict the future. When downtime is detected, we do not yet know if the primary has actually failed, if the failure can self-heal quickly, or if a failed health-check was actually due to temporary network issues or a faulty monitor, etc. We also cannot know how long it would take for a backup to be ready to process requests from clients—depending on the disk state when the primary failed, it could be seconds or minutes.

But suppose an *oracle* did know, at the moment of apparent failure, how long recovery would take on a backup as well as how long the primary’s apparent failure would last. Such an oracle could achieve considerably better availability by avoiding timeouts completely while avoiding a slow recovery when the primary’s failure is temporary. In particular, the oracle’s optimal decision would be to choose the shorter of waiting for the primary to self-heal or immediately beginning recovery on a backup without waiting.

We propose a new failover design, *speculative recovery*, that makes similarly optimal choices *in practice*, without knowing the future. Speculative recovery pursues both paths (Figure 1) in parallel and either aborts recovery if the primary becomes available first, or irreversibly marks the primary failed if recovery completes first.

To accomplish this, speculative recovery creates and attaches an independent clone of the primary’s disk to a new

backup instance immediately when primary downtime is detected. The backup begins recovery from the cloned disk, potentially in parallel with the primary’s continued operation if it is not actually failed. This results in a “superposition” where the parent and child disks are permitted to temporarily diverge, as long as neither is observed externally. Once one of them is observed (i.e., if the primary becomes available or when clients are redirected to a fully recovered backup), the superposition collapses and the unobserved disk is destroyed. Specifically, this superposition state collapses in two cases:

- **Observing the primary.** If any *writes* to the parent disk are observed, the primary is assumed to be available and the superposition is collapsed by aborting recovery on the backup and deallocating the child disk.
- **Observing the backup.** If recovery on the backup completes successfully and no writes have been issued to the parent disk, the superposition is collapsed by deallocating the parent disk, destroying the primary, and promoting the backup to be the new primary by pointing all clients to it.

Thus, speculative recovery on the backup can complete though the primary *may* still be operational, while guaranteeing external correctness. As long as the application is crash-consistent, observing clients cannot distinguish between speculative recovery and REDS, except that failover may appear much faster. If a client receives an acknowledgement from the primary for a request that modifies application state, crash consistency mandates that the primary must have written to the disk, which would halt failover to the backup, in turn ensuring the backup is never observable.

Conversely, if a client is directed to communicate with the backup, the primary cannot have acknowledged any state-modifying operations or is no longer servicing client requests, and thus the backup’s state is consistent with all previous reads from the primary.

To realize these important properties, speculative recovery introduces two new disaggregated storage primitives: `super` and `collapse`. `super` produces a temporal, performant disk clone using copy-on-write (COW) semantics, resulting in a superposition in which the parent disk (attached to the primary) and the child disk (attached to the backup) diverge from the same state. `collapse` destroys the parent disk if and only if it has not changed since `super`, otherwise it destroys the, yet unobserved, child disk.

It is critical that disk clones spawned by `super` are fast to create and performant, so as not to slow down recovery on the backup significantly. `super` uses a new form of COW disk, *collocated-clone*, that improves COW writes over existing designs by up to an order of magnitude and performs almost as well as a regular, non-COW disk. Similarly, `collapse` must operate atomically—it must determine whether any writes have been made to the parent disk *and* block future writes if

not, atomically—but should also not unduly delay failover. `collapse` uses a single, global *dirty bit* for the entire parent disk to track whether writes have occurred in the superposition, allowing `collapse` to use a simple protocol with only a single round trip to one storage shard. In both cases, these designs are enabled by the temporal nature of the superposition.

4 Design

This section details our design for speculative recovery. It describes the system components, the design of `super`, the design of `collapse`, why and when speculative recovery is correct, and finally discusses some performance concerns.

4.1 Components and Overview

A speculative recovery system consists of three components: (1) an instance pool to host applications; (2) disaggregated storage that provides highly durable and highly available virtual disks to applications with the `super` and `collapse` primitives; (3) a failure monitor that monitors the health of the running application instances and coordinates speculative recovery for failed instances.

When the failure monitor presumes the primary instance is unhealthy, e.g., if the monitor fails to connect to the application, it initiates speculative recovery. It invokes `super` on the primary’s disaggregated disk which creates a lightweight clone using COW semantics (§4.2). In addition, `super` causes the parent disk to begin tracking writes to support the `collapse` protocol (§4.3). Next, the monitor spins up a new backup instance from the same application boot image as the primary, except with the cloned child disk attached in place of the parent. When the backup finishes restarting the application, the monitor calls `collapse`, which either atomically promotes the backup if there have been no writes to the parent disk or deallocates it if there have been writes.

4.2 `super`: Creating a Disk Superposition

As the backup instance boots and starts up the application, it may write to the child disk. For example, `fsck` might fix corruption in the file system and the application may replay and commit or rollback uncommitted transactions from its write-ahead log (WAL). Meanwhile, the primary is still allowed to function should it become available before recovery on the backup is complete. As a result, the parent and child disks are likely to diverge. However, this divergence retains application correctness because the backup is not observable to clients until after it is determined that the primary has not acknowledged any state-modifying requests.

This design is relatively simple to realize using existing primitives in disaggregated disks. In particular, many disaggregated disks provide copy-on-write clones that are quick to create. In principle, this should allow speculative recovery

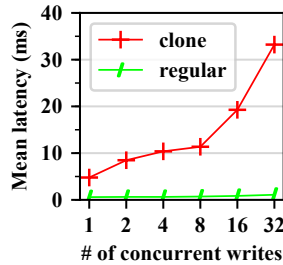


Figure 2: **Concurrent writes on EBS.** Writes are issued simultaneously in batches of 1 write to 32 concurrent writes.

to explore both paths simultaneously—waiting out the unresponsiveness on the primary and recovering the application on the backup—and achieve the same outcome as an oracle. In addition, COW clones provide the child disk with the same level of durability guarantee as the parent since dirtied data blocks are copied as new blocks and thus can be applied the same replication schema (e.g., three-way replication).

Unfortunately, existing designs for COW disk clones perform very poorly for recovery workloads. We conducted black-box experiments on EBS to measure the I/O performance of EBS clones. EBS supports clones by first creating a snapshot from a volume and then creating a new volume from that snapshot. Figure 2 shows the write performance of EBS clones with varying levels of parallelism. Normally, concurrent writes on a regular EBS volume can exploit disk parallelism well (the green line): the average latency when 32 writes are in-flight is only 2.6x the latency of a single write. However, for a cloned EBS volume (the red line), this relation becomes 7x, indicating significant performance bottlenecks for a cloned volume under highly parallel writes.

To understand the underlying reasons, we instrumented the open-source Ceph codebase where a similar behavior exists: on a cloned disk, the average latency with 32 writes in-flight is 7.1x the latency of a single write (more results and details are described in §6.2). We discovered two fundamental problems with the Ceph clone implementation, and we speculate that these may be general to many other clone designs.

First, because disaggregated disks typically treat a clone’s dirtied blocks like any other new disk block, most COW designs copy dirtied blocks to different storage shards than the ones hosting the original blocks. This results in considerable overhead compared to modifying blocks in place. Second, each dirtied block requires allocating a new location in the storage area network, which is typically a blocking operation. As a result, concurrent writes that touch mostly newly dirtied blocks are performed in *sequence* rather than in parallel.

In short, copying dirty blocks to new locations over the network increases single write latency significantly, while serialized allocation eliminates most of the parallelism benefit for concurrent writes. These overheads are reasonable for typical uses of COW-cloned disks, where COW writes, and particularly concurrent writes, are infrequent [25]. How-

ever, a recovery workload is often write-intensive. As a result, these overheads can dramatically increase the time to recover applications—in some cases from seconds on a regular disaggregated disk to several minutes on a COW clone.

4.2.1 Collocated-Clone

super addresses both performance issues, copying overhead and serialization of COW writes, using a mechanism we term *collocated-clone*. Rather than treating copied dirty blocks the same as newly allocated blocks, *collocated-clone* reuses the parent’s allocation table to collocate child blocks with their corresponding parent blocks. This accomplishes two things. First, copying a dirtied block *never* traverses the network, as child blocks are always on the same shard as the parent blocks. Second, COW writes never require a blocking allocation operation as the parent’s allocation table already contains enough information to derive the child block’s location—specifically, it is always on the same shard as the parent’s and its name can be derived from the parent block’s name.

As a result, COW writes in *collocated-clone* require only marginally more work than normal writes. The dirtied block must be copied, but only locally—incurring local disk overhead, but not network overhead. Moreover, these writes *never* require a new block allocation, so concurrent writes are always just as parallelizable as on a regular, non-COW disk.

Collocated-clone is not suitable for many uses of COW-clones because it risks amplifying any skew in the original disk’s allocation. However, in speculative recovery, clones are temporary: after a short period of coexistence, it is either the child being deallocated or the child succeeding the parent and carrying on. Shards only need to have sufficient extra storage to store dirtied blocks temporarily.

In addition, *collocated-clone* only provides limited isolation between the parent and child. Because *collocated-clone* does not require the parent to do COW, the parent can directly update its data blocks in case it self-recovers. Thus, if the parent updates a data block the child has not copied, the child can see those updates, breaking the isolation. Again, this is permissible in the special semantics of *superposition* since if the parent is ever updated, the child will never be externalized.

4.3 collapse: Collapsing a Superposition

By allowing the parent and child disks to diverge in their *superposition*, speculative recovery introduces potential application inconsistency that must be hidden from clients. To prevent such inconsistencies, *collapse* uses a single disk-global dirty bit to indicate whether there have been writes applied to the parent disk since the creation of the child. It must also have a means of atomically promoting the backup instance to be the new primary, even with in-flight operations from the old primary.

Tracking primary writes. When `super` is invoked on a disk, its disk-global `dirty` and `allow-write` bits are initially set to false and true, respectively, on a fault-tolerant tracking shard in the storage cluster (this may simply be one of the data shards). When a shard of the parent disk receives a write request, before servicing the write, it requests permission to perform the write from the tracking shard. If the `allow-write` bit is true, the tracking shard sets the `dirty` bit and allows the shard to proceed with the write. Otherwise it responds that the shard should reject the write.

Atomic promotion. `collapse` operations are performed on the tracking shard. This shard atomically checks the `dirty` bit and, if it is still false, sets the `allow-write` bit to false, preventing any future write attempts to the parent disk. It then returns an acknowledgment that the parent disk is disabled and being deallocated. Otherwise, if the `dirty` bit is true, it responds that the parent disk has been observed, that the backup should be taken down, and begins asynchronously deallocating the child disk, aborting failover.

Tracking disk modification using a disk-global `dirty` bit allows `collapse` to complete quickly, as the only atomic operation is limited to a single node in the disaggregated storage cluster, avoiding expensive multi-node protocols such as two-phase commit. Such a tracking mechanism may be unnecessary and inappropriate for long-lived COW-clones that may have subsequent children and grandchildren. However, due to the ephemeral nature of the superposition, and because it is at most one clone of a disk at any given time, this design allows `collapse` to be supported efficiently.

After promotion of the backup is complete, the primary might still be able to service client reads from its in-memory cache, even though its disk has been deallocated by `collapse`. This would externalize potentially stale values. To prevent this, a stronger method is needed to sever the old primary from the clients. The specific mechanisms to achieve this may be cloud-platform dependent, but one option is to use an “elastic IP” [18] to remap the old primary’s IP address to the newly promoted primary, automatically rerouting clients. Other mechanisms such as using a firewall to block the primary’s access to the network would also work.

4.4 Correctness and the Failure Model

Speculative recovery ensures correctness, i.e., linearizability [37], by ensuring two properties. First, only one instance of an application is accessible to clients at any point in time. Second, if the backup is promoted and becomes accessible, its state begins from the previous primary’s last acknowledged changes and thus it looks like a continuation of the old primary. The first property is achieved trivially using atomic promotion. The second property is achieved using `super` and `collapse` in sequence for a crash-consistent application: a backup is only promoted by `collapse` if there have been no

writes since `super`, and thus the disk it recovers from must include the previous primary’s last acknowledged changes as required by crash consistency.

Our design of `collapse` uses writes to the primary’s disk as a signal of liveness to abort failover. This will correctly detect crash failures where the primary stops completely. However, it will not detect more nuanced kinds of failures such as partial failures or fail-slow failures. For example, even if the primary is disconnected from the clients, it may still write to disk for internal operations such as log rotation and garbage collection. This means that writes to the primary’s disk may not always reflect client-visible application state changes, and `collapse` would abort failover in these cases. In addition, fail-slow failures, where applications are slow but not inaccessible, can occur [26, 34, 40, 60]. In these failure situations, speculative recovery can be falsely and repeatedly aborted, causing an increased failover latency. In all these cases, speculative recovery should fall back to REDS by using a timeout to force failover when recovery is aborted repeatedly.

5 Implementation

We implemented a prototype speculative recovery system, SpecREDS, and deployed it on AWS EC2. The instance pool is implemented as a docker container pool on top of EC2 compute instances, where application images can be directly pulled from the docker registry. The failure monitor is implemented as a simple daemon process that pings the application instances with read-only queries to determine connectivity and health. As an independent component, the monitor also needs to be fault-tolerant. Many orchestration architectures provide fault-tolerant monitors such as those in EC2 Auto Scaling groups, Kubernetes, etc.

Our implementation of the disaggregated storage layer is based on Ceph [68], an open-source distributed storage system. On top of its backend object store called RADOS, Ceph provides highly durable and highly available block storage called `rbd` (Rados Block Device) that can remotely attach a `rbd` disk as a Linux block device through its kernel driver. SpecREDS focuses on the block interface due to its prevalent adoption on cloud and its simpler interface. We believe that the concept of speculative recovery can be applied to other cloud storage interfaces like network file systems [17] and object stores [19]. The implementation is based on Ceph release v16.2.4. The artifact of SpecREDS is publicly available. Please refer to the appendix for the artifact description.

Background on Ceph `rbd`. We give a short background on `rbd` necessary to understand our implementation. `rbd` also provides disk clone functionality: a disk snapshot is first taken, then a disk clone can be created from that snapshot. While clone creation is fast, `rbd`’s native clone implementation has the performance problems of copying over the network and serialized concurrent COW writes, as discussed in §4.2.

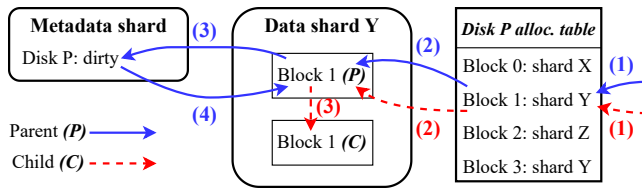


Figure 3: **Parent and child write path after `super`.** The parent and child disks are assigned ID P and C , respectively. Only the first parent write to shard Y performs steps 3 and 4.

`rbd` implements the functionality of an “allocation table” with two separate utilities. First, each `rbd` disk has an object map, a bit map indicating the *existence* of the disk’s data objects. A COW write needs to update the child’s object map by marking the corresponding bit “dirty”, which is a blocking operation due to locking, causing the effect of serialized concurrent COW writes. Second, the location of a data object is *calculated* deterministically by an algorithm based on the object name and the cluster layout [69]. Since objects have unique names, a child object will likely be placed on a different shard than its parent. For clarity, this section assumes that a `rbd` disk has an “allocation table” that combines the two utilities, as shown in Figure 3.

In addition, a `rbd` clone disk depends on its parent snapshot, and such dependency prevents the parent from deletion unless the cloned child is deleted first. As a result, for repeated failovers, the latest child will carry a chain of parent dependencies. These parents keep taking up space even though they are not needed anymore, as well as the child keeps suffering from COW penalties even after the failover is complete.

Collocation by reusing parent’s “allocation table.” To accomplish this, `super` directly assigns the parent disk to the child, including all data objects and the object map. This achieves two things. First, a COW write *never* needs to update the object map since by reusing parent’s object map, the corresponding bit is already updated by the parent. Second, the child uses the same object names as the parent to locate objects, allowing for collocation of parent and child objects.

To differentiate, the parent and the child are assigned a unique ID. When accessing the disk, they identify themselves to the storage cluster using that ID. This means that creating the child disk is fast because it only involves the assignment of a unique ID. The names of the objects are tagged with the unique ID to identify the object ownership (parent or child).

To determine how to serve a child I/O, the data shard first checks the existence of the child object and the corresponding parent object by directly querying the backend object store. COW is performed for a child write if the child object does not exist but the parent object does. Figure 3 demonstrates this process (dashed red arrows). By reusing the allocation table, child access will be directed to the same shard holding the corresponding parent objects (steps 1 and 2) and thus allowing for collocation (step 3).

Object size. Another factor affecting COW performance is the object size since objects are the minimal unit of copying. But if a COW write contains some whole objects, copying is unnecessary for these objects. With large objects, data copying imposes huge overhead; with small objects, writes are more likely to contain whole objects to reduce copying overhead, but the overhead of allocating more smaller objects could overwhelm and thus degrade the overall performance. Our benchmark shows that `rbd`’s default 4 MB object size is not ideal for many database applications whose default page size is only 4 KB to 64 KB. The the sweet spot for these applications is around 64 KB.

Dirty bit tracking and atomic promotion. `collapse` elects the data shard that stores the parent disk’s metadata (which is a single object) as the tracking shard. When `super` is invoked on the parent disk, its unique ID is registered to the tracking shard and then broadcasted to all data shards. The data shards then add the ID to a tracking list. When receiving a write with ID in the tracking list, the data shard must ask the tracking shard for permission to proceed (blue solid arrow, step 3 in figure 3). If permission is granted, the data shard can then submit this write and remove the ID from the tracking list; otherwise, this write must be rejected.

The tracking shard, by default, grants permission to any data shard requesting (step 4 in figure 3), sets the dirty bit associated with the ID, and notifies the other data shards to remove the ID from their tracking lists. The tracking shard also persists the dirty bit by writing it to the disk’s metadata, allowing it to be replicated along with the metadata. In case the current tracking shard fails, another shard holding a replica of the metadata is elected the new tracking shard.

When initiating an atomic promotion, based on the dirty bit status of the parent disk, the tracking shard performs either of the two actions *atomically*: (1) if the dirty bit is set, the tracking shard returns an error to indicate that promotion is rejected and the child disk should be deallocated; (2) otherwise, the tracking shard starts rejecting all requests-for-permission to the parent disk and acknowledges that promotion can proceed.

Dirty bit tracking adds one additional RTT to at most the number of writes equivalent to the number of data shards in the cluster. Our evaluation shows that this has negligible performance impact (§6.2).

Deallocation with garbage collection. `collapse` deallocates the child disk by asynchronously garbage-collecting all objects associated with the child’s unique ID. Similarly, the parent disk is deallocated by asynchronously garbage-collecting parent objects that have a corresponding child object and reassigning those who do not to the child’s ownership. After this process is complete, the child no longer depends on the parent and no longer needs to do COW. Asynchronous garbage collection minimizes the performance impact to the storage cluster’s normal operation.

6 Evaluation

This evaluation answers the following questions:

- How does the performance of colocated-clone disks compare to that of normal disks and general-clone disks? (§6.2)
- What is the recovery latency for various applications and failure scenarios when using a colocated-clone disk compared to using a normal disk and a general-clone disk? (§6.3)
- How does the failover latency of SpecREDS compare to REDS? (§6.4)
- What are the overheads of SpecREDS over REDS in terms of application performance after recovery, resource overhead, and overhead due to false positives? (§6.5)

We find that our implementation of a colocated-clone disk provides disk-level performance *close* to that of a normal disk and is much faster than a general-clone disk (§6.2). This performance translates to recovery latency when using a colocated-clone disk being close to using a normal disk (§6.3). This similar recovery latency leads to speculative recovery always providing failover latency comparable to REDS and often providing much lower failover latency across a wide variety of failover scenarios (§6.4).

6.1 Experimental Setup

We conducted our evaluation on EC2. The SpecREDS storage layer has four storage shards by EC2 instance type `i3en` with access to 7500 GB NVMe local SSDs and 25 Gbps network bandwidth. As shown in Table 1, our storage layer delivers performance comparable to popular cloud storage services.

For the primary and backup instances, we use the `m5n` instance type with 16 vCPUs, 64 GB RAM, and 25 Gbps network bandwidth, and for the application clients, we use an instance with 32 vCPUs, 128 GB RAM, and 25 Gbps network bandwidth. All instances are in the same availability zone as each other and the storage layer. We also set up a simple docker orchestrator environment on the primary and backup instances where applications are running in docker containers. The client instance runs `oltpbench` [31] with 100 virtual clients sending requests to the active instance. The primary is initially the active instance, while the failover process with our orchestrator makes the backup the active instance as it completes. The failure monitor, which pings the instances every second, runs as a separate daemon process on the client machine. We believe that this setup mimics existing systems like EC2 Auto Scaling groups and GCP Kubernetes Engine.

We pick three representative database applications: MySQL with InnoDB, PostgreSQL, and MariaDB with RocksDB. These applications meet the requirements of REDS and are widely used. The `oltpbench` client loads these application by running the TPC-C workload [65].

	KIOPS	Tput (MB/s)	Latency (ms)
EBS gp3	16/16	1000/1000	0.5/0.7
GCP SSD PD	15/15	245/245	0.6/0.7
Our storage layer	75/26	1000/630	0.38/2.0

Table 1: **Raw disk performance.** Comparing the raw disk performance of EBS General Purpose SSD (gp3), GCP SSD Persistent Disk, and our storage layer. Numbers in each cell are for read/write.

6.2 Disk-level Performance

To build up to an end-to-end availability comparison, we start by showing a disk-level performance comparison between a regular, non-COW `rbd` disk, a colocated-clone disk implemented with `super`, and a general-clone disk implemented with native `rbd` cloning (`rbd-clone`). We use an object size of 64 KB for all disks. The experiments examine single write performance, concurrent write performance, performance for real recovery workloads, and the impact of the dirty bit on the parent disk performance. Our results indicate that the main source of improvement comes from the elimination of object map update operations, which could increase the latency of a single write by 6.1x under highly parallel I/Os.

Single COW write latency. As the first set of experiments, we isolate the latency impact of the COW designs when there is no concurrency with an experiment that issues *single writes*, where only a single write is in flight at a time. Figure 4a compares the mean latency (averaged over 20,000 writes) of COW writes with `super` and `rbd-clone` to normal writes with `rbd` for varying write sizes. A closed-loop client issues writes to random offsets.

For writes smaller than the object size, write latency on `super` is 14% higher than `rbd` while `rbd-clone` is 220% higher. `super` provides this similar performance because it avoids an object map update operation and does the copy locally instead of having to transmit data over the network. When the write size equals the object size (64 KB), no COW is necessary. This isolates the latency effect of object map update operations. For `rbd-clone`, this results in 2 ms of added latency (the update operation is basically another write), while `super` has identical performance to normal writes because it does not need to update the object map.

Concurrent COW writes. Next, we evaluate the latency of COW write under varying levels of concurrency. Figure 4b shows the mean latency of 4 KB COW writes as we vary concurrency from 1 write in flight at a time up to 32 writes in flight. A closed-loop client simultaneously issues n writes to random offsets, waits for all of their responses, and then repeats this process.

The latency of `rbd-clone` increases sharply with concurrency: the mean latency with 32 writes in flight is 6.1x higher than the mean latency with 1 write in flight. We found that this high latency is due to parallel object map update opera-

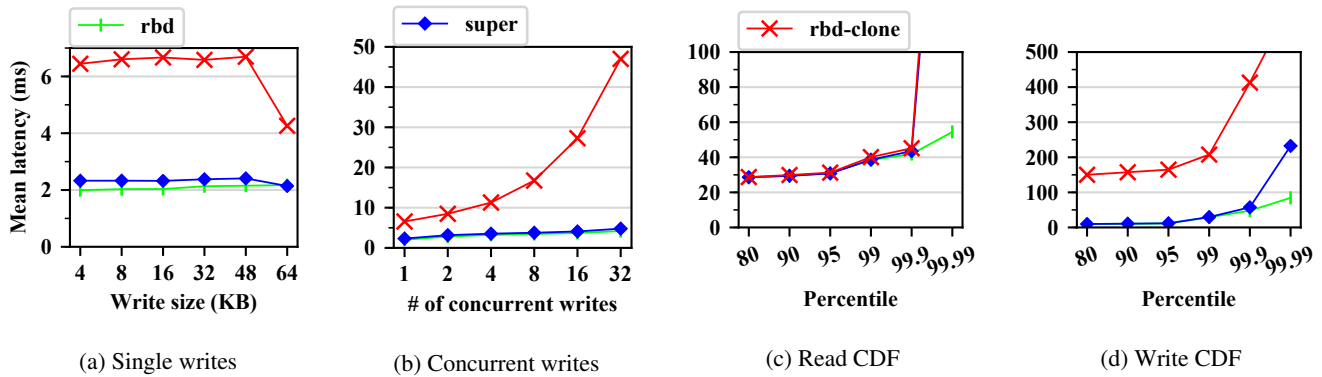


Figure 4: **COW performance comparison.** (a) Latency for varying size COW writes with no concurrency; (b) Latency for 4 KB COW writes under increasing levels of concurrency; (c)/(d) Read/write CDFs from replaying a trace of recovery operations.

tions being serialized by the client’s disk driver. In contrast, `super` provides similar performance to `rd` under concurrency because it avoids updates to the object map: both have comparable mean latency that ranges from 2 ms with a concurrency of 1 to 4 ms with a concurrency of 32.

Performance on real recovery workloads. To quantify how these improvements of `super` translate to performance in real application recovery workloads, Figures 4c and 4d show read and write CDFs collected from replaying a recovery workload trace with `fiio` [32]. The trace captures the recovery work for a Postgres database with 20 GB TPCC data and 1 GB of WAL at the time of an injected kernel panic failure.

For read, all three disks have similar read latency up to p99.9. `super` and `rd-clone` have higher read latency beyond p99.9 due to the overhead of COW that may occupy a majority of disk throughput under high load and cause heavy I/O contention. For write, the write latency of `rd-clone` is much higher than `rd`. In contrast, the write latency of `super` is comparable to `rd` up to p99.9.

Dirty bit tracking overhead. As discussed in §5, the dirty bit tracking mechanism of SpecREDS may impact the performance of the parent disk because some parent writes require an additional round trip to set the dirty bit. This could become a problem if the primary instance self-heals and continues serving the application. We performed an experiment that invokes `super` every second while measuring raw IOPS on the parent disk. Even under such an extreme condition, the parent disk achieves the same IOPS numbers. Thus, the only overhead of dirty bit tracking is increased latency for the few writes that set the dirty bit when the primary is still alive.

Summary. We believe that these disk-level improvements of `super`, as shown in Figure 4, can achieve recovery latency very close to a regular `rd` disk in real failure scenarios, enabling end-to-end application availability improvement for SpecREDS, as presented in the next two subsections.

6.3 Application Recovery Latency

We ran a series of experiments to understand how disk-level performance of the three disk types (`rd`, `super`, and `rd-clone`) affects recovery latency as we vary failure type and failure timing. SpecREDS operates on a disk clone (`super` by default or `rd-clone`) with COW penalties, which increases application recovery latency compared to REDS using a regular `rd` disk without COW. It is critical for such latency increase to be relatively minor to show practical improvement in end-to-end application availability (§6.4)

For these experiments, the application initially runs in a container on the primary instance, handling requests from the clients. Then, a failure is injected to the primary. To isolate recovery latency, the failure monitor detects loss of connectivity with no timeout and immediately initiates failover, and the application restarts on the backup instance. The recovery latency is measured at the client side as the length of time between when the TPC-C throughput drops to zero and when it resumes. Failures are injected either by synchronously stopping the docker container and unmounting the disk (clean failures) or by causing a kernel panic (unclean failures).

We found the type of failure has a major effect on recovery latency. Stopping the primary container tries to gracefully shut down the application (this is the case for MariaDB but not for MySQL and Postgres), and unmounting the disk flushes file system cache such that the file system is not corrupted. Therefore, the disk is in a cleaner state and can recover faster. Kernel panic, on the other hand, immediately crashes the instance without giving a chance to clean up, leaving the disk in an unclean state that takes longer for the backup to recover. In addition, we found the size of WAL at time of failure also significantly impacts recovery latency.

Our full range of experiments have recovery latencies that vary from 1–70 s when run on `rd`. We capture block-level traces of those recovery workloads with `blktrace` and then replay them with `fiio` on `rd`, `super`, and `rd-clone`. Replaying traces ensures the workload is identical for all three disks.

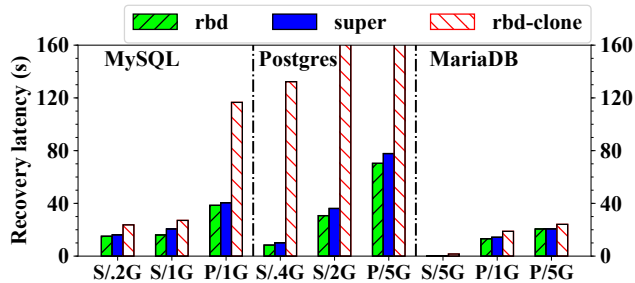


Figure 5: **Application recovery latency from various disk states.** Recovery latency is shown for our three applications running on `rd`, `super`, and `rd-clone`. Failures are injected using docker stop (S) or a kernel panic (P). Labels are failure types followed by WAL size in GB.

To make results legible while demonstrating the effect of varying WAL sizes and failure types, we select three scenarios to show for each application. The recovery latency for each disk with these scenarios is shown in Figure 5. In all cases, we see that `super` improves performance over `rd-clone`. This is especially pronounced for Postgres whose recovery workload is generally more write-intensive, exacerbating the write performance bottlenecks in `rd-clone` shown in §6.2. Further, recovery on `super` is only slightly slower than recovery on `rd` by 13% on average.

6.4 End-to-end Failover Latency

To quantify the effect of speculative recovery for complete end-to-end failover scenarios, we simulated various failover scenarios and compare the latency across REDS (using `rd`), SpecREDS (using `super` by default), SpecREDS (using `rd-clone`), and the oracle model (using `rd`). The oracle model shows the lower bound on failover latency: it runs recovery on a `rd` disk immediately after a primary issues its last write (or simply waits for primary to come back online, whichever is shorter). Thus, the oracle shows failover latency without either REDS’s timeout or SpecREDS’s slower disk performance. On the other hand, REDS initiates recovery after a full timeout, while SpecREDS initiates much sooner after only *one second* of an unresponsive ping.

The simulations explored three variables: the primary-is-failed timeout, the recovery latency for the backup, and if/when the primary self-heals. Results are divided into broad categories depending on the timeout length (short, medium, or long), recovery length (short or long), and whether the primary self-heals after the timeout but before backup recovery completes (true or false positive recovery for REDS). Results with a long timeout (e.g., the Kubernetes default timeout of five minutes) are similar to a medium timeout but have even higher failover latency for REDS, so we only show results for a medium timeout. Results with false positive recovery for

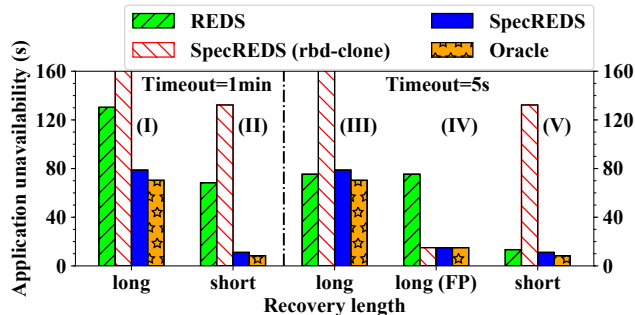


Figure 6: **End-to-end failover latency.** Representative failover scenarios, picked by varying the lengths of timeout and recovery. Bar group IV shows a false positive (FP) failover for REDS

REDS all similarly inflate only the latency of REDS, so we only show one of these results. This leads to five categories.

Figure 6 shows a representative result from each of these five categories. The medium timeout is one minute and the short timeout is five seconds. The recovery latencies are picked from the results in §6.3. The long recovery is from Postgres with an unclear failure and a 5 GB WAL (around 70 seconds of recovery latency on `rd`). The short recovery is from Postgres with a clean failure and a 0.4 GB WAL (around 8 seconds on `rd`).

The two leftmost bar groups show scenarios with medium timeouts and demonstrate one major part of SpecREDS’s availability improvement over REDS. Because SpecREDS starts recovery early without waiting for a full timeout, it completes failover much sooner and thus significantly reduces application unavailability.

The three rightmost bars in Figure 6 demonstrate short timeout failure scenarios. Bar groups III and V shows similar performance for REDS and SpecREDS with a long (III) or short (V) recovery. In these cases, SpecREDS start recovery slightly sooner than REDS. But, its recovery takes slightly longer because its `super` disk is slightly slower than the `rd` disk used by REDS. With a long recovery (III), this makes REDS’s unavailability marginally shorter than SpecREDS. With a short recovery (V), this makes SpecREDS’s unavailability marginally shorter than REDS. Finally, bar group IV shows a false positive failover where the primary is available again (we used 15 seconds for illustration) shortly after the timeout. SpecREDS decreases unavailability considerably in this scenario by allowing the primary to continue instead of committing to recovery on the backup with no turning back.

Overall, there are three takeaways. First, the failover latency of SpecREDS (`rd-clone`) is consistently the highest, indicating that the improved performance of the `super` disk is the key to achieving the availability improvement of SpecREDS. Second, SpecREDS achieves significantly lower failover latency when REDS uses a medium timeout (bar groups I and

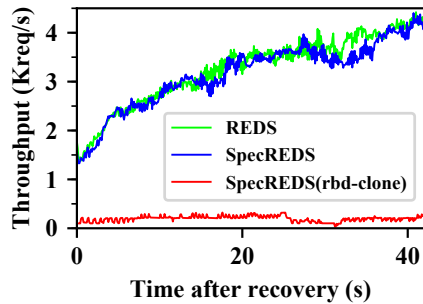


Figure 7: **Throughput after recovery.** Time 0 is right after recovery completes and clients resume.

II) because this timeout dominates REDS’s unavailability; SpecREDS also achieves lower failover latency for false positives when REDS uses a short timeout (IV), while achieving similar failover latency in other cases (III and V). Third, SpecREDS is always close to the oracle lower bound, suggesting it achieves most of the possible availability improvement for a REDS-based fault tolerance scheme.

6.5 Other SpecREDS Overheads

To understand the other overheads of SpecREDS, we evaluated application performance immediately after recovery is complete, analyzed production health monitor logs to estimate the resource overhead of SpecREDS, and discussed the performance overhead on the storage layer’s normal operation due to false positives.

Application performance after recovery. After the backup instance completes recovery and gets promoted, collapse asynchronously transfers parent objects to the child. During this time, COW is still used for writes that go to objects whose ownership has not yet been transferred. Figure 7 compares application performance following recovery (Postgres, unclean failure with 1 GB WAL) on REDS, SpecREDS, and SpecREDS (`rbd-clone`). SpecREDS using `rbd-clone` has low throughput due to the continued impact of COW because the parent-child dependency still exists, as discussed in §5. In contrast, we see that SpecREDS has a throughput curve very similar to REDS. Thus, we conclude that SpecREDS adds negligible overhead to application performance after recovery.

Resource overhead of SpecREDS. Due to running two instances concurrently during speculative recovery and the possibility of aborted recovery, SpecREDS incurs additional resource overhead compared to REDS. The key to understanding SpecREDS’s resource overhead is to see how often it would be incurred. We analyzed a complete collection of health monitor logs from more than 80 production caching servers for the past five years. On average, a server is reported

inaccessible once every 2.8 days. Of these reported events, 90% are transient: the server becomes accessible again within 10 seconds. Even with such a high false positive rate, the frequency of possible server inaccessible event is quite small. The resource overhead of SpecREDS would be, on average, an unnecessary backup instance allocation for up to 10 seconds once every 3.1 days: a 0.004% overhead.

Performance overhead due to false positives. False positives that trigger speculative failover that is aborted impose performance overhead on the storage layer due to garbage collection (GC). This may be troublesome since short-lived failures are common in today’s data center networks [21, 48, 54, 60], which could introduce frequent GC that could harm the storage layer’s normal operation. Our log analysis described above, however, found that a server is reported inaccessible once every 2.8 days on average, meaning that SpecREDS incurs GC overhead only once every few days. Moreover, GC incurs minor performance overhead, since GC is asynchronous and thus does not block regular disk I/O operations, as shown in Figure 7.

7 Related Work

This section reviews related work on application-level replication, state machine replication, virtual machine replication, slow recovery in databases, shared storage clustering, disk snapshotting, and other related uses of speculation within systems. The most closely related work is the industry’s adoption of REDS, which is introduced in §2.2 and discussed extensively throughout the paper.

Application-level replication. This is a widely implemented technique for providing high-availability fault tolerance. SQL databases, including MySQL, PostgreSQL, Microsoft SQL Server, as well as NoSQL databases such as MongoDB, replicate client transactions synchronously and persistently to backups before responding to clients [7–10]. This can provide excellent performance with failover latency shorter than SpecREDS. However, it requires an extensive implementation for each individual application since the replication logic and implementation are application-specific. Many useful persistence applications do not provide high-availability at the application layer, including SQLite, LevelDB, and RocksDB [1, 3, 4, 6, 11]. In contrast, Both REDS and SpecREDS support these applications without any modification or explicit support, since they work at the block-device layer.

Application-level replication requires multiple application instances at all times to provide fault tolerance: at least the primary instance and one backup. In contrast, REDS and SpecREDS only run a single instance almost at all times, which makes it far cheaper.

Application-level replication may also have lower performance in normal operation since it runs expensive replication protocols for client requests. We believe that this argument

needs meticulous measurements to validate because REDS and SpecREDS do not eliminate the need for a replication protocol but instead runs it at the storage level. In addition, the replicas in application-level replication can provide read-only throughput. Though disaggregated storage can also offer better disk-level read throughput from data replicas, single application instance often cannot fully utilize it due to bottlenecks at CPU and network bandwidth [41]

State machine replication (SMR). This technique provides high availability for applications that use its interface, which is typically a log of requests executed in order [62]. SMR is typically implemented either using a consensus algorithm like Paxos [44] or a primary-backup approach [24]. SMR can often provide shorter recovery times than SpecREDS. But, like application-level replication, it requires multiple instances and thus is more costly than SpecREDS.

Virtual machine (VM) replication. This technique provides application-agnostic high-availability fault tolerance [23, 29, 53, 63]. This technique replicates an entire VM and thus can make any application or a collection of applications fault tolerant. However, VM replication is heavy-weight because it replicates the entire virtual machine (e.g., all changes to memory must be replicated before they are externalized to provide linearizability). Also, it requires at least two instances at all times to provide fault tolerance. Speculative recovery supports the smaller set of applications that are crash consistent, but is much lighter weight and provides high availability at a much lower cost

Database slow recovery. This is a technique that precedes the cloud by decades where logs are periodically shipped to a backup that stores, but does not apply, them until a failover is needed. This similarly requires fewer backup resources in the normal case but results in slower recovery. REDS and SpecREDS build on this technique to provide a similar tradeoff more generally for any crash-consistent application and in a cloud-native way by using disaggregated storage to provide the backup its own copy of the disk instead of requiring any computation from a backup.

Shared storage clustering. This technique allows a storage volume to be attached to and accessible from multiple application instances at the same time, enabling faster failover in a clustered application setup without dismounting and remounting the volume to another instance [51]. The cloud-native version of this technique is “multi-attach” [14]. These techniques require a standby backup instance, which is not the case for REDS and SpecREDS.

Snapshots and checkpoints. Other forms of storage copy such as snapshots and checkpoints are widely used for data backup and rollback-based disaster recovery [38, 46, 67]. Many cloud platforms also support automatically taking snapshots of application disks on a user-specified schedule. How-

ever, this method does not provide linearizability amid failures because updates following the latest snapshot will be lost.

Speculation. This is a widely used technique to accelerate the performance of systems. Here we discuss a few of these systems that inspired us. Zyzzyva [42] is a Byzantine fault tolerance SMR protocol where the replicas speculatively execute client requests without agreeing on a single total ordering, and it is then the client’s responsibility to observe and help resolve any inconsistencies. Speculative recovery adopts a similar idea that inconsistency can be allowed temporarily and resolved later.

Speculative Paxos [59] is a SMR protocol where replicas speculatively execute client requests based on the message delivery order provided by the underlying network layer. In cases where this order is violated, a reconciliation protocol is in place to rollback inconsistent operations. Such inconsistencies are detected before externalizing. Speculative recovery is similar in that inconsistencies cannot be externalized. This is also inspired by a similar idea in “rethink the sync” [55] where external clients are the real observer of the system.

Speculation is also widely adopted for tolerating tail latency in data-parallel computing such as Hadoop and Spark [64, 70]. When a computing job is taking an unexpectedly long time, the same job will be sent to another worker, and the system uses the results from whichever finishes first. Hedged requests are a similar technique that is used for applications that access many backend systems [30] as well as other domains such as RAID storage arrays [35, 36]. Speculative recovery is similar to these techniques in that there are two racing paths and latency is determined by the first path to finish.

8 Conclusions

We presented speculative recovery, a cheap, highly available fault-tolerance scheme based on disaggregated storage for crash-consistent applications. At the core of speculative recovery are the two new primitives, `super` and `collapse`, for disaggregated storage. `super` provides performant disk clones with the novel colocated-clone design, and `collapse` ensures application correctness, i.e., linearizability, in a failover process with a disk-global dirty bit. Speculative recovery achieves the same level of resource efficiency as REDS with significantly higher availability in most failover scenarios.

Acknowledgments

We thank our anonymous shepherd and reviewers for their many constructive comments. We thank Khiem Ngo and Jeffrey Helt for their helpful discussions. We thank Cloud-Lab [61] for providing compute resources used in the development of this project. This material is based upon work supported by the National Science Foundation under Grants No. 1763546, 2028869, and 2106530.

References

- [1] About SQLite. <https://www.sqlite.org/about.html>.
- [2] Docker. <https://www.docker.com/>.
- [3] How we use RocksDB at Rockset. <https://rockset.com/blog/how-we-use-rocksdb-at-rockset/>.
- [4] LevelDB Store. <https://activemq.apache.org/leveldb-store>.
- [5] Linux Containers. <https://linuxcontainers.org/>.
- [6] Litereplica: Replication Support for SQLite. <http://litereplica.io/sqlite-replication.html>.
- [7] Microsoft SQL Server Replication. <https://docs.microsoft.com/en-us/sql/relational-databases/replication/sql-server-replication?view=sql-server-ver15>.
- [8] MongoDB Replication. <https://docs.mongodb.com/manual/replication/>.
- [9] MySQL Replication. <https://dev.mysql.com/doc/refman/8.0/en/replication.html>.
- [10] PostgreSQL Replication. <https://www.postgresql.org/docs/9.2/runtime-config-replication.html>.
- [11] rocksplicator, RocksDB Replication. <https://github.com/pinterest/rocksplicator>.
- [12] StatefulSets – Kubernetes. <https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/>.
- [13] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.
- [14] Amazon. Attach a volume to multiple instances with Amazon EBS Multi-Attach. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volumes-multi.html>.
- [15] Amazon. EC2 Auto Scaling groups. <https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroup.html>.
- [16] Amazon. Elastic Block Storage. <https://aws.amazon.com/ebs>.
- [17] Amazon. Elastic File System. <https://aws.amazon.com/efs/>.
- [18] Amazon. Elastic IP addresses. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html>.
- [19] Amazon. Simple Storage Service (S3). <https://aws.amazon.com/s3/>.
- [20] Michael Baentsch, Georg Molter, and Peter Sturm. Introducing Application-Level Replication and Naming into Today's Web. *Computer Networks and ISDN Systems*, 28(7–11):921–930, May 1996.
- [21] Peter Bailis and Kyle Kingsbury. The network is reliable: An informal survey of real-world communications failures. *Queue*, 12(7):20–32, 2014.
- [22] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 13–24, New York, NY, USA, 2005. Association for Computing Machinery.
- [23] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault Tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 1–11, New York, NY, USA, 1995. Association for Computing Machinery.
- [24] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. Distributed systems. *ch. The Primary-Backup Approach*, pages 199–216, 1993.
- [25] Ceph. rbd Persistent Read-only Cache. <https://docs.ceph.com/en/latest/rbd/rbd-persistent-read-only-cache/>.
- [26] Mike Y. Chen, Anthony Accardi, and Dave Patterson. Path-Based Failure and Evolution Management. In *First Symposium on Networked Systems Design and Implementation*, NSDI '04, San Francisco, CA, March 2004. USENIX Association.
- [27] Vijay Chidambaram, Thanumalayan Sankaranarayanan Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 228–243, New York, NY, USA, 2013. Association for Computing Machinery.
- [28] Brian Cho and Ergin Seyfe. Taking Advantage of a Disaggregated Storage and Compute Architecture. In *Spark+AI Summit 2019*, SAIS '19, April 2019.

- [29] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, San Francisco, CA, April 2008. USENIX Association.
- [30] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, February 2013.
- [31] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.
- [32] fio. Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html.
- [33] Google. GCP Persistent Disks. <https://cloud.google.com/persistent-disk>.
- [34] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Gollhofer, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Biral Runesha, Mingzhe Hao, and Huaicheng Li. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *16th USENIX Conference on File and Storage Technologies*, FAST '18, pages 1–14, Oakland, CA, February 2018. USENIX Association.
- [35] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 168–183, New York, NY, USA, 2017. Association for Computing Machinery.
- [36] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 173–190. USENIX Association, November 2020.
- [37] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
- [38] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, feb 1988.
- [39] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions. In *2018 USENIX Annual Technical Conference*, USENIX ATC '18, pages 879–891, Boston, MA, July 2018. USENIX Association.
- [40] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 150–155, New York, NY, USA, 2017. Association for Computing Machinery.
- [41] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.
- [42] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 45–58, New York, NY, USA, 2007. Association for Computing Machinery.
- [43] Kubernetes. kube-controller-manager. <https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/>.
- [44] Leslie Lamport. Paxos Made Simple, Fast, and Byzantine. In *Proceedings of the 6th International Conference on Principles of Distributed Systems*, OPODIS '02, pages 7–9, 2002.
- [45] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanael Cherié, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony Rowstron. Understanding Rack-Scale Disaggregated Storage. In *9th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '17, Santa Clara, CA, July 2017. USENIX Association.
- [46] LVM-HOWTO. Taking a Backup Using Snapshots. https://tldp.org/HOWTO/LVM-HOWTO/snapshots_backup.html.

- [47] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.
- [48] Shicong Meng, Arun K. Iyengar, Isabelle M. Rouvellou, Ling Liu, Kisung Lee, Balaji Palanisamy, and Yuzhe Tang. Reliable State Monitoring in Cloud Datacenters. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing, CLOUD '12*, pages 951–958, 2012.
- [49] Microsoft. Azure blob storage. <https://azure.microsoft.com/en-us/services/storage/blobs>.
- [50] Microsoft. High availability in Azure Database for PostgreSQL – Single Server. <https://docs.microsoft.com/en-us/azure/postgresql/concepts-high-availability>.
- [51] Microsoft. Use Cluster Shared Volumes in a failover cluster. <https://docs.microsoft.com/en-us/windows-server/failover-clustering/failover-cluster-csvs>.
- [52] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference, SIGCOMM '21*, page 106–122, New York, NY, USA, 2021. Association for Computing Machinery.
- [53] Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Abounaga, Kenneth Salem, and Andrew Warfield. RemusDB: Transparent High Availability for Database Systems. *Proceedings of the VLDB Endowment*, 4(11):738–748, August 2011.
- [54] Srihari Nelakuditi, Sanghwan Lee, Yinzhe Yu, Zhi-Li Zhang, and Chen-Nee Chuah. Fast Local Rerouting for Handling Transient Link Failures. *IEEE/ACM Transactions on Networking*, 15(2):359–372, April 2007.
- [55] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the Sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06*, page 1–14, USA, 2006. USENIX Association.
- [56] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. Rabia: Simplifying State-Machine Replication Through Randomization. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21*, page 472–487, New York, NY, USA, 2021. Association for Computing Machinery.
- [57] Colin Percival. EC2 boot time benchmarking. <https://www.daemonology.net/blog/2021-08-12-EC2-boot-time-benchmarking.html>.
- [58] Thanumalayan Sankaranarayanan Pillai, Vijay Chidambaram, Ramnathan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14*, pages 433–448, Broomfield, CO, October 2014. USENIX Association.
- [59] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI '15*, pages 43–57, Oakland, CA, May 2015. USENIX Association.
- [60] Rahul Potharaju and Navendu Jain. When the Network Crumbles: An Empirical Study of Cloud Network Failures and Their Impact on Services. In *Proceedings of the 4th Annual Symposium on Cloud Computing, SoCC '13*, New York, NY, USA, 2013. Association for Computing Machinery.
- [61] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *login USENIX Magazine*, 39(6), 2014.
- [62] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
- [63] Rahul Singh, David Irwin, Prashant Shenoy, and K.K. Ramakrishnan. Yank: Enabling Green Data Centers to Pull the Plug. In *10th USENIX Symposium on Networked Systems Design and Implementation, NSDI '13*, pages 143–155, Lombard, IL, April 2013. USENIX Association.
- [64] Riza O. Suminto, Cesar A. Stuardo, Alexandra Clark, Huan Ke, Tanakorn Leesatapornwongsa, Bo Fu, Daniar H. Kurniawan, Vincentius Martin, Maheswara

Rao G. Uma, and Haryadi S. Gunawi. PBSE: A Robust Path-Based Speculative Execution for Degraded-Network Tail Tolerance in Data-Parallel Frameworks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 295–308, New York, NY, USA, 2017. Association for Computing Machinery.

- [65] TPC-C. An On-Line Transaction Processing Benchmark. <http://www.tpc.org/tpcc/>.
- [66] Peng Wang, Kaiyuan Zhang, Rong Chen, Haibo Chen, and Haibing Guan. Replication-Based Fault-Tolerance for Large-Scale Graph Processing. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 562–573, 2014.
- [67] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and Steven Hand. Parallax: Managing storage for a million machines. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems*, HotOS '05, page 4, USA, 2005. USENIX Association.
- [68] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 307–320, USA, 2006. USENIX Association.
- [69] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, pages 31–31, 2006.
- [70] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '08, page 29–42, USA, 2008. USENIX Association.

A Artifact Appendix

Abstract

The artifact provides a framework for evaluating SpecREDS as shown in the evaluation section of the paper. The artifact includes the source code of SpecREDS's disaggregated storage layer (based on Ceph), configuration files and pre-captured application recovery traces, and handy scripts for instrumenting the experiments. Readers can easily use this artifact to reproduce figures shown in the paper.

Scope

There are two main claims from the paper that the artifact seeks to validate: **(1)** the disk-level I/O performance of `super`, our novel design and implementation of light-weight, fast disk clones, is *close* to that of a regular, non-COW disk, while significantly outperforming Ceph's existing clone implementation `rbd-clone`; **(2)** SpecREDS using `super` can bring practical end-to-end application availability improvement over REDS in various failover scenarios.

Specifically, the paper uses Figures 4 and 5 to prove the point of the first claim and Figures 6 and 7 to prove the second claim. The artifact contains experiments to reproduce these four figures, and readers should be able to compare them with the original figures in the paper to validate the claims.

Contents

The artifact contains the following items

- The source code of SpecREDS's disaggregated storage
- A simple tool for measuring disk-level performance
- Pre-configured configs, disk images, and traces
- Scripts for instrumenting all experiments
- Detailed readmes

Hosting

The artifact is hosted on our public GitHub repository at <https://github.com/princeton-sns/specreds>. The tag for the OSDI/ATC artifact evaluation is `atc22ae`. To get started, please follow the detailed instructions in the repo.

Requirements

The artifact does not require special hardware or software, but we highly recommend running the artifact on CloudLab with a `c220g2` or `c220g5` machine where the artifact is tested to be reproducible. If not available, we recommend using a machine with at least 16 CPU cores, 64 GB memory, 400 GB of free space on an SSD, and Ubuntu 20.04. We also provide a `qcow2` image for booting up a QEMU VM.

Direct Access, High-Performance Memory Disaggregation with DIRECTCXL

Donghyun Gouk, Sangwon Lee, Miryeong Kwon, Myoungsoo Jung
Computer Architecture and Memory Systems Laboratory,
Korea Advanced Institute of Science and Technology (KAIST)
<http://camelab.org>

Abstract

New cache coherent interconnects such as CXL have recently attracted great attention thanks to their excellent hardware heterogeneity management and resource disaggregation capabilities. Even though there is yet no real product or platform integrating CXL into memory disaggregation, it is expected to make memory resources practically and efficiently disaggregated much better than ever before.

In this paper, we propose directly accessible memory disaggregation, DIRECTCXL that straight connects a host processor complex and remote memory resources over CXL's memory protocol (CXL.mem). To this end, we explore a practical design for CXL-based memory disaggregation and make it real. As there is no operating system that supports CXL, we also offer CXL software runtime that allows users to utilize the underlying disaggregated memory resources via sheer load/store instructions. Since DIRECTCXL does not require any data copies between the host memory and remote memory, it can expose the true performance of remote-side disaggregated memory resources to the users.

1 Introduction

Memory disaggregation has attracted great attention thanks to its high memory utilization, transparent elasticity, and resource management efficiency [1–3]. Many studies have explored various software and hardware approaches to realize memory disaggregation and put significant efforts into making it practical in large-scale systems [4–16].

We can broadly classify the existing memory disaggregation runtimes into two different approaches based on how they manage data between a host and memory server(s): i) page-based and ii) object-based. The page-based approach [4–10] utilizes virtual memory techniques to use disaggregated memory without a code change. It swaps page cache data residing on the host's local DRAMs from/to the remote memory systems over a network in cases of a page fault. On the other hand, the object-based approach handles disaggregated memory from a remote using their own database such as a key-value store instead of leveraging the virtual memory systems [11–16]. This approach can address the challenges imposed by address translation (e.g., page faults, context switching, and write amplification), but it requires significant source-level modifications and interface changes.

While there are many variants, all the existing approaches need to move data from the remote memory to the host memory over remote direct memory access (RDMA) [4, 5, 11–13, 15, 16] (or similar fine-grain network interfaces [7, 9, 10, 17]). In addition, they even require managing locally cached data in either the host or memory nodes. Unfortunately, the data movement and its accompanying operations (e.g., page cache management) introduce redundant memory copies and software fabric intervention, which makes the latency of disaggregated memory longer than that of local DRAM accesses by multiple orders of magnitude. In this work, we advocate *compute express link* (CXL [18]), which is a new concept of open industry standard interconnects offering high-performance connectivity among multiple host processors, hardware accelerators, and I/O devices [19]. CXL is originally designed to achieve the excellency of heterogeneity management across different processor complexes, but both industry and academia anticipate its cache coherence ability can help improve memory utilization and alleviate memory over-provisioning with low latency [20–22]. Even though CXL exhibits a great potential to realize memory disaggregation with low monetary cost and high performance, it has not been yet made for production, and there is no platform to integrate memory into a memory pooling network.

We demonstrate DIRECTCXL, direct accessible disaggregated memory that connects host processor complex and remote memory resources over CXL's memory protocol (CXL.mem). To this end, we explore a practical design for CXL-based memory disaggregation and make it real. Specifically, we first show how to disaggregate memory over CXL and integrate the disaggregated memory into processor-side system memory. This includes implementing CXL controller that employs multiple DRAM modules on a remote side. We then prototype a set of network infrastructure components such as a CXL switch in order to make the disaggregated memory connected to the host in a scalable manner. As there is no operating system that support CXL, we also offer CXL software runtime that allows users to utilize the underlying disaggregated memory resources through sheer load/store instructions. DIRECTCXL does not require any data copies between the host memory and remote memory, and therefore, it can expose the true performance of remote-side disaggregated memory resources to the users.

In this work, we prototype DIRECTCXL using many cus-

tomized memory add-in-cards, 16nm FPGA-based processor nodes, a switch, and a PCIe backplane. On the other hand, DIRECTCXL software runtime is implemented based on Linux 5.13. To the best of our knowledge, this is the first work that brings CXL 2.0 into a real system and analyzes the performance characteristics of CXL-enabled disaggregated memory design. The results of our real system evaluation show that the disaggregated memory resources of DIRECTCXL can exhibit DRAM-like performance when the workload can enjoy the host processor’s cache. When the load/store instructions go through the CXL network and are served from the disaggregated memory, DIRECTCXL’s latency is shorter than the best latency of RDMA by $6.2\times$, on average. For real-world applications, DIRECTCXL exhibits $3\times$ better performance than RDMA-based memory disaggregation, on average.

2 Memory Disaggregation and Related Work

2.1 Remote Direct Memory Access

The basic idea of memory disaggregation is to connect a host with one or more memory nodes, such that it does not restrict a given job execution because of limited local memory space. For the backend network control, most disaggregation work employ remote direct memory access (RDMA) [4, 5, 11–13, 15, 16] or similar customized DMA protocols [7, 9, 10]. Figure 1 shows how RDMA-style data transfers (one-sided RDMA) work. For both the host and memory node sides, RDMA needs hardware support such as RDMA NIC (RNIC [23]), which is designed toward removing the intervention of the network software stack as much as possible. To move data between them, processes on each side first require defining one or more memory regions (MRs) and letting the MR(s) to the underlying RNIC. During this time, the RNIC driver checks all physical addresses associated with the MR’s pages and registers them to RNIC’s memory translation table (MTT). Since those two RNICs also exchange their MR’s virtual address at the initialization, the host can simply send the memory node’s destination virtual address with data for a write. The remote node then translates the address by referring to its MTT and copies the incoming data to the target location of MR. Reads over RDMA can also be performed in a similar manner. Note that, in addition to the memory copy operations (for DMA), each side’s application needs to prepare or retrieve the data into/from MRs for the data transfers, introducing additional data copies within their local DRAM [24].

2.2 Swap: Page-based Memory Pool

Page-based memory disaggregation [4–10] achieves memory elasticity by relying on virtual memory systems. Specifically, this approach intercepts paging requests when there is a page fault, and then it swaps the data to a remote memory node instead of the underlying storage. To this end, a disaggregation driver underneath the host’s kernel swap daemon (*kswapd*) converts the incoming block address to the memory node’s

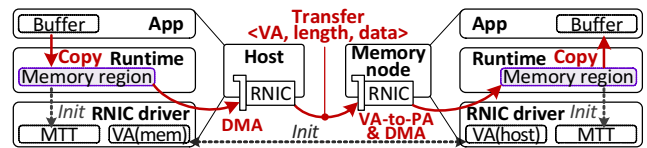


Figure 1: Data movement over RDMA.

virtual address. It then copies the target page to RNIC’s MR and issues the corresponding RDMA request to the memory node. Since all operations for memory disaggregation is managed under *kswapd*, it is easy-to-adopt and transparent to all user applications. However, page-based systems suffer from performance degradation due to the overhead of page fault handling, I/O amplifications, and context switching when there are excessive requests for the remote memory [16].

Note that there are several studies that migrate locally cached data in a finer granular manner [4–7] or reduce the page fault overhead by offloading memory management (including page cache coherence) to the network [8] or memory nodes [9, 10]. However, all these approaches use RDMA (or a similar network protocol), which is essential to cache the data and pay the cost of memory operations for network handling.

2.3 KVS: Object-based Memory Pool

In contrast, object-based memory disaggregation systems [11–16] directly intervene in RDMA data transfers using their own database such as key-value store (KVS). Object-based systems create two MRs for both host and memory node sides, each dealing with buffer data and submission/completion queues (SQ/CQ). Generally, they employ a KV hash-table whose entries point to corresponding (remote) memory objects. Whenever there is a request of Put (or Get) from an application, the systems place the corresponding value into the host’s buffer MR and submit it by writing the remote side of SQ MR over RDMA. Since the memory node keeps polling SQ MR, it can recognize the request. The memory node then reads the host’s buffer MR, copies the value to its buffer MR over RDMA, and completes the request by writing the host’s CQ MR. As it does not lean on virtual memory systems, object-based systems can address the overhead imposed by page swap. However, the performance of object-based systems varies based on the semantics of applications compared to page-based systems; *kswapd* fully utilizes local page caches, but KVS does not for remote accesses. In addition, this approach is unfortunately limited because it requires significant source-level modifications for legacy applications.

3 Direct Accessible Memory Aggregation

While caching pages and network-based data exchange are essential in the current technologies, they can unfortunately significantly deteriorate the performance of memory disaggregation. DIRECTCXL instead directly connects remote memory resources to the host’s computing complex and allows users to access them through sheer load/store instructions.

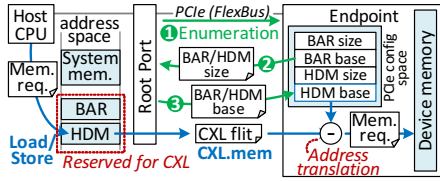


Figure 2: DIRECTCXL's connection method.

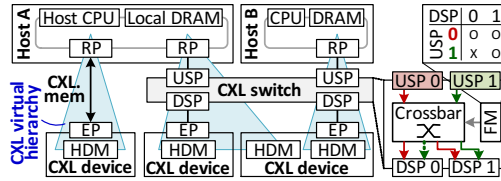


Figure 3: DIRECTCXL's network and switch.

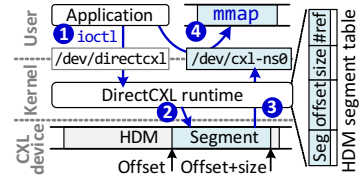


Figure 4: DIRECTCXL software runtime.

3.1 Connecting Host and Memory over CXL

CXL devices and controllers. In practice, existing memory disaggregation techniques still require computing resources at the remote memory node side. This is because all DRAM modules and their interfaces are designed as passive peripherals, which require the control computing resources. CXL.mem in contrast allows the host computing resources directly access the underlying memory through PCIe buses (*FlexBus*); it works similar to local DRAM, connected to their system buses. Thus, we design and implement CXL devices as pure passive modules, each being able to have many DRAM DIMMs with its own hardware controllers. Our CXL device employs multiple DRAM controllers, connecting DRAM DIMMs over the conventional DDR interfaces. Its CXL controller then exposes the internal DRAM modules to FlexBus through many PCIe lanes. In the current architecture, the device's CXL controller parses incoming PCIe-based CXL packets, called *CXL flits*, converts their information (address and length) to DRAM requests, and serves them from the underlying DRAMs using the DRAM controllers.

Integrating devices into system memory. Figure 2 shows how CXL devices' internal DRAMs are mapped (exposed) to a host's memory space over CXL. The host CPU's system bus contains one or more CXL root ports (*RP*s), which connect one or more CXL devices as endpoint (*EP*) devices. Our host-side kernel driver first enumerates CXL devices by querying the size of their base address register (*BAR*) and their internal memory, called host-managed device memory (*HDM*), through PCIe transactions. Based on the retrieved sizes, the kernel driver maps *BAR* and *HDM* in the host's reserved system memory space and lets the underlying CXL devices know where their *BAR* and *HDM* (base addresses) are mapped in the host's system memory. When the host CPU accesses an *HDM* system memory through load/store instruction, the request is delivered to the corresponding *RP*, and the *RP* converts the requests to a CXL flit. Since *HDM* is mapped to a different location of the system memory, the memory address space of *HDM* is different from that of *EP*'s internal DRAMs. Thus, the CXL controller translates the incoming addresses by simply deducting *HDM*'s base address from them and issues the translated request to the underlying DRAM controllers. The results are returned to the host via a CXL switch and FlexBus. Note that, since *HDM* accesses have no software intervention or memory data copies, DIRECTCXL can expose the CXL device's memory resources to the host with low access latency.

Designing CXL network switch. Figure 3a illustrates how DIRECTCXL can disaggregate memory resources from a host using one or more and CXL devices, and Figure 3b shows our CXL switch organization therein. The host's CXL *RP* is connected to *upstream port* (*USP*) of either a CXL switch or the CXL device directly. The CXL switch's *downstream port* (*DSP*) also connects either another CXL switch's *USP* or the CXL device. Note that our CXL switch employs multiple *USPs* and *DSPs*. By setting an internal routing table, our CXL switch's *fabric manager* (*FM*) reconfigures the switch's crossbar to connect each *USP* to a different *DSP*, which creates a virtual hierarchy from a root (host) to a terminal (CXL device). Since a CXL device can employ one or more controllers and many DRAMs, it can also define multiple logical devices, each exposing its own *HDM* to a host. Thus, different hosts can be connected to a CXL switch and a CXL device. Note that each CXL virtual hierarchy only offers the path from one to another to ensure that no host is sharing an *HDM*.

3.2 Software Runtime for DirectCXL

In contrast to RDMA, once a virtual hierarchy is established between a host and CXL device(s), applications running on the host can directly access the CXL device by referring to *HDM*'s memory space. However, it requires software runtime/driver to manage the underlying CXL devices and expose their *HDM* in the application's memory space. We thus support DIRECTCXL runtime that simply splits the address space of *HDM* into multiple segments, called *cxl-namespace*. DIRECTCXL runtime then allows the applications to access each CXL-namespace as memory-mapped files (*mmap*).

Figure 4 shows the software stack of our runtime and how the application can use the disaggregated memory through *cxl-namespaces*. When a CXL device is detected (at a PCIe enumeration time), DIRECTCXL driver creates an entry device (e.g., `/dev/directcxl`) to allow users to manage a *cxl-namespace* via *ioctl*. If users ask a *cxl-namespace* to `/dev/directcxl`, the driver checks a (physically) contiguous address space on an *HDM* by referring to its *HDM* segment table whose entry includes a segment's offset, size, and reference count (recording how many *cxl-namespaces* that indicate this segment). Since multiple processes can access this table, its header also keeps necessary information such as spinlock, read/write locks, and a summary of table entries (e.g., valid entry numbers). Once DIRECTCXL driver allocates a segment based on the user request, it creates a device for *mmap* (e.g., `/dev/cxl-ns0`) and updates the segment table. The user

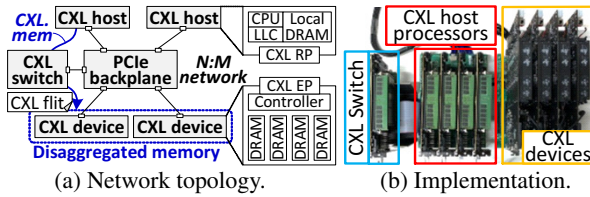


Figure 5: CXL-enabled cluster.

application can then map the `cxl-namespace` to its process virtual memory space using `mmap` with `vm_area_struct`.

Note that `DIRECTCXL` software runtime is designed for direct access of CXL devices, which is a similar concept to the memory-mapped file management of persistent memory development toolkit (PMDK [25]). However, it is much simpler and more flexible for namespace management than PMDK. For example, PMDK’s namespace is very much the same idea as NVMe namespace, managed by file systems or DAX with a fixed size [26]. In contrast, our `cxl-namespace` is more similar to the conventional memory segment, which is directly exposed to the application without a file system employment.

3.3 Prototype Implementation

Figure 5a illustrates our design of a CXL network topology to disaggregate memory resources, and the corresponding implementation in a real system is shown in Figure 5b. There are n numbers of compute hosts connected to m number of CXL devices through a CXL switch; in our prototype, n and m are four, but those numbers can scale by having more CXL switches. Specifically, each CXL device prototype is built on our customized add-in-card (AIC) CXL memory blade that employs 16nm FPGA and 8 different DDR4 DRAM modules (64GB). In the FPGA, we fabricate a CXL controller and eight DRAM controllers, each managing the CXL endpoint and internal DRAM channels. As yet there is no processor architecture supporting CXL, we also build our own in-house host processor using RISC-V ISAs, which employs four out-of-order cores whose last-level cache (LLC) implements CXL RP. Each CXL-enabled host processor is implemented in a high-performance datacenter accelerator card, taking a role of a host, which can individually run Linux 5.13 and `DIRECTCXL` software runtime. We expose four CXL devices (32 DRAM modules) to the four hosts through our PCIe backplane. We extended the backplane with one more accelerator card that implements `DIRECTCXL`’s CXL switch. This switch implements FM that can create multiple virtual hierarchies, each connecting a host and a CXL device in a flexible manner.

To the best of our knowledge, there are no commercialized CXL 2.0 IPs for the processor side’s CXL engines and CXL switch. Thus, we built all `DIRECTCXL` IPs from the ground. The host-side processors require advanced configuration and power interface (ACPI [27]) for CXL 2.0 enumeration (e.g., RP location and RP’s reserved address space). Since RISC-V does not support ACPI yet, we enable the CXL enumeration by adding such information into the device tree [28]. Specifically, we update an MMIO register designated as a property of

the tree’s node to let the processor know where CXL RP exists. On the other hand, we add a new field (`cxl-reserved-area`) in the node to indicate where an HDM can be mapped. Our in-house softcore processors work at 100MHz while CXL and PCIe IPs (RP, EP, and Switch) operate at 250MHz.

4 Evaluation

Testbed prototypes for memory disaggregation. In addition to the CXL environment that we implemented in Section 3.3 (`DirectCXL`), we set up the same configuration with it for our RDMA-enabled hardware system (RDMA). For RDMA, we use Mellanox ConnectX-3 VPI InfiniBand RNIC (56Gbps, [29]) instead of our CXL switch as RDMA network interface card (RNIC). In addition, we port Mellanox OpenFabric Enterprise Distribution (OFED) v4.9 [30] as an RDMA driver to enable RNIC in our evaluation testbed. Lastly, we port FastSwap [1] and HERD [12] into RISC-V Linux 5.13.19 computing environment atop RDMA, each realizing page-based disaggregation (Swap) and object-based disaggregation (KVS).

For better comparison, we also configure the host processors to use only their local DRAM (Local) by disabling all the CXL memory nodes. Note that we used the same testbed hardware mentioned above for both CXL experiments and non-CXL experiments but differently configured the testbed for each reference. For example, our testbed’s FPGA chips for the host (in-house) processors and CXL devices use all the same architecture/technology and product line-up.

Benchmark and workloads. Since there is no microbenchmark that we can compare different memory pooling technologies (RDMA vs. `DirectCXL`), we also build an in-house memory benchmark for in-depth analysis of those two technologies (Section 4.1). For RDMA, this benchmark allocates a large size of the memory pool at the remote side in advance. This benchmark allows a host processor to send random memory requests to a remote node with varying lengths; the remote node serves the requests using the pre-allocated memory pool. For `DirectCXL` and Local, the benchmark maps `cxl-namespace` or `anonymous mmap` to user spaces, respectively. The benchmark then generates a group of RISC-V memory instructions, which can cover a given address length in a random pattern and directly issues them without software intervention. For the real workloads, we use Facebook’s deep learning recommendation model (DLRM [31]), an in-memory database used for the HERD evaluation (MemDB [12]), and four graph analysis workloads (MIS [32], BFS [33], CC [34], and BC [35]) coming from Ligra [36]. All their tables and data structures are stored in the remote node, while each host’s local memory handles the execution code and static data. Table 1 summarizes the per-node memory usage and total data sizes for each workload that we tested.

4.1 In-depth Analysis of RDMA and CXL

In this subsection, we compare the performance of RDMA and CXL technologies when the host and memory nodes are

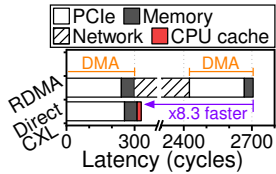


Figure 6: RDMA vs. CXL.

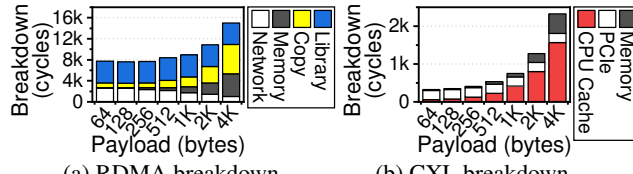


Figure 7: Sensitivity tests.

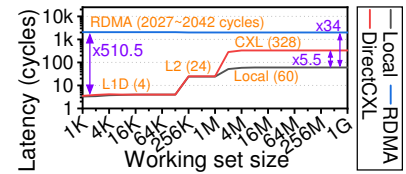


Figure 8: Memory hierarchy performance.

configured through a 1:1 connection. Figure 6 shows latency breakdown of RDMA and DirectCXL when reading 64 bytes of data. One can observe from the figure that RDMA requires two DMA operations, which doubles the PCIe transfer and memory access latency. In addition, the communication overhead of InfiniBand (Network) takes 78.7% (2129 cycles) of the total latency (2705 cycles). In contrast, DirectCXL only takes 328 cycles for memory load request, which is $8.3\times$ faster than RDMA. There are two reasons behind this performance difference. First, DirectCXL straight connects the compute nodes and memory nodes using PCIe while RDMA requires protocol/interface changes between InfiniBand and PCIe. Second, DirectCXL can translate memory load/store request from LLC into the CXL flits whereas RDMA must use DMA to read/write data from/to memory.

Sensitivity tests. Figure 7a decomposes RDMA latency into essential hardware (Memory and Network), software (Library), and data transfer latencies (Copy). In this evaluation, we instrument two user-level InfiniBand libraries, `libibverbs` and `libmlx4` to measure the software side latency. Library is the primary performance bottleneck in RDMA when the size of payloads is smaller than 1KB (4158 cycles, on average). As the payloads increase, Copy gets longer and reaches 28.9% of total execution time. This is because users must copy all their data into RNIC’s MR, which takes extra overhead in RDMA. On the other hand, Memory and Network shows a performance trend similar to RDMA analyzed in Figure 6. Note that the actual times of Network (Figure 7a) do not decrease as the payload increases; while Memory increases to handle large size of data, RNIC can simultaneously transmit the data to the underlying network. These overlapped cycles are counted by Memory in our analysis. As shown in Figure 7b, the breakdown analysis for DirectCXL shows a completely different story; there is neither software nor data copy overhead. As the payloads increase, the dominant component of DirectCXL’s latency is LLC (CPU Cache). This is because LLC can handle 16 concurrent misses through miss status holding registers (MSHR) in our custom CPU. Thus, many memory requests (64B) composing a large payload data can be stalled at CPU, which takes 67% of the total latency to handle 4KB payloads. PCIe shown in Figure 7a does not decrease as the payloads increase because of a similar reason of RDMA’s Network. However, it

	Per-node usage		Total usage	Data stored in remote memory
	Local	Remote		
DLRM [31]	Less than 100MB	17GB	68GB	Embedding tables.
MemDB [12]	100MB	4GB	16GB	Key-value pairs and tree structure.
Ligra [36]		7GB	28GB	Deserialized graph structure.

Table 1: Memory usage characteristic of each workload.

is not as much as what Network did as only 16 concurrent misses can be overlapped. Note that PCIe shown in Figures 6 and 7b includes the latency of CXL IPs (RP, EP, and Switch), which is different from the pure cycles of PCIe physical bus. The pure cycles of PCIe physical bus (FlexBus) account for 28% of DirectCXL latency. The detailed latency decomposition will be analyzed in Section 4.2.

Memory hierarchy performance. Figure 8 shows latency cycles of different components in the system’s memory hierarchy. While Local and DirectCXL exhibits CPU cache by lowering the memory access latency to 4 cycles, RDMA has negligible impacts on CPU cache as their network overhead is much higher than that of Local. The best-case performance of RDMA was 2027 cycles, which is $6.2\times$ and $510.5\times$ slower than that of DirectCXL and L1 cache, respectively. DirectCXL requires 328 cycles whereas Local requires only 60 cycles in the case of L2 misses. Note that the performance bottleneck of DirectCXL is PCIe including CXL IPs (77.8% of the total latency). This can be accelerated by increasing the working frequency, which will be discussed shortly.

4.2 Latency Distribution and Scaling Study

Latency distribution. In addition to the latency trend (average) we reported above, we also analyze complete latency behaviors of Local, RDMA, and DirectCXL. Figure 9 shows the latency CDF of memory accesses (64B) for the different pooling methods. RDMA shows the performance curve, which ranges from 1790 cycles to 4006 cycles. The reason why there is a difference between the minimum and maximum latency of RDMA is RNIC’s MTT memory buffer and CPU caches for data transfers. While RDMA cannot take the benefits from direct load/store instruction with CPU caches, its data transfers themselves utilize CPU caches. Nevertheless, RDMA cannot avoid the network accesses for remote memory accesses, making its latency worse than Local by $36.8\times$, on average. In contrast, the latency behaviors of DirectCXL are similar to Local. Even though the latency of DirectCXL (reported in Figures 6 and 7b) is the average value, its best performance is the same as Local (4~24 cycles). This is because, as we showed in the previous section, DirectCXL can take the benefits of CPU caches directly. The tail latency is $2.8\times$ worse than Local, but its latency curve is similar to that of Local. This is because both DirectCXL and Local use the same DRAM (and there is no network access overhead).

Speed scaling estimation. The cycle numbers that we reported here are measured at each host’s CPU using register-level instrumentation. We believe it is sufficient and better

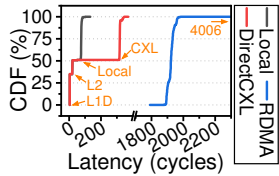


Figure 9: Memory-level latency CDF (64B).

Measurement clock domain →	DIRECTCXL	PCIe 5.0 x8 (Estimated)	
	CPU (100MHz)	CPU (1.2GHz)	Time delay
L1/L2 cache	30	30	25 ns
CXL IPs (2.0)*	165	287	239 ns
PCIe FlexBus	91	69	57 ns
DRAM controller	42	126	105 ns
Total	328	512	426 ns

*Including RP, EP, and Switch

Unit: cycles

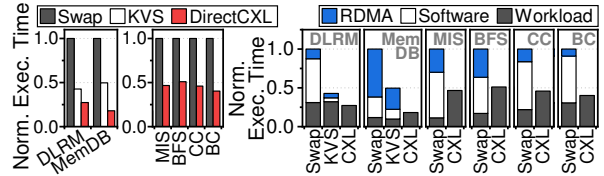
Table 2: Latency breakdown and estimated 64B load latency.

than a cross-time-domain analysis to decompose the system latency. Nevertheless, we estimate a time delay in cases where the target system accelerates the frequency of its processor complex and CXL IPs (RP, EP, and Switch) by 1.2GHz and 1GHz, respectively. Table 2 decomposes `DirectCXL`'s latency of a 64B memory load and compares it with the estimated time delay. The cycle counts of L1/L2 cache misses are not different as they work in all the same clock domain of CPU. While other components (FlexBus, CXL IPs, and DRAM controller) speed up by $4\times$ ($250\text{MHz} \rightarrow 1\text{GHz}$), the number of cycles increases since CPU gets faster by $12\times$. Note that, as the version of PCIe is changed and the number of lanes for PCIe increases by double, FlexBus's cycles decrease. The table includes the time delays corresponding to the estimated system from the CPU's viewpoint. While the time delay of FlexBus is pretty good ($\sim 60\text{ns}$), the corresponding CXL IPs have room to improve further with a higher working frequency.

4.3 Performance of Real Workloads

Figure 10a shows the execution latency of `Swap`, `KVS`, and `DirectCXL` when running `DLRM`, `MemDB`, and four workloads from `Ligra`. For better understanding, all the results in this subsection are normalized to those of `Swap`. For `Ligra`, we only compare `DirectCXL` with `Swap` because `Ligra`'s graph processing engines (handling in-/out-edges and vertices) is not compatible with a key-value structure. `KVS` can reduce the latency of `Swap` as it addresses the overhead imposed by page-based I/O granularity to access the remote memory. However, it has two major issues behind `KVS`. First, it requires significant modification of the application's source codes, which is often unable to service (e.g., `MIS`, `BFS`, `CC`, `BC`). Second, `KVS` requires heavy computation such as hashing at the memory node, which increases monetary costs. In contrast, `DirectCXL` without having a source modification and remote-side resource exhibits $3\times$ and $2.2\times$ better performance than `Swap` and even `KVS`, respectively.

To better understand this performance improvement of `DirectCXL`, we also decompose the execution times into `RDMA`, network library intervention (`Software`), and application execution itself (`Workload`) latencies, and the results are shown in Figure 10b. This figure demonstrates where `Swap` degrades the overall performance from its execution; 51.8% of the execution time is consumed by kernel swap daemon (`kswapd`) and `FastSwap` driver, on average. This is because `Swap` just expands memory with the local and remote based on `LRU`, which makes its page exchange frequent. The



(a) Execution Time.

(b) Execution breakdown.

Figure 10: Real workload performance.

reason why `KVS` shows performance better than `Swap` in the cases of `DLRM` and `MemDB` is mainly related to workload characteristics and its service optimization. For `DLRM`, `KVS` loads the exact size of embeddings rather than a page, which reduces `Swap`'s data transfer overhead as high as $6.9\times$. While `KVS` shows the low overhead in our evaluation, `RDMA` and `Software` can linearly increase as the number of inferences increases; in our case, we only used 13.5MB (0.0008%) of embeddings for single inference. For `MemDB`, as `KVS` stores all key-value pairs into local DRAM, it only accesses remote-side DRAM to inquiry values. However, it spends 55.3% and 24.9% of the execution time for `RDMA` and `Software` to handle the remote DRAMs, respectively. In contrast, `DirectCXL` removes such hardware and software overhead, which exhibits much better performance than `Swap` and `KVS`. Note that `MemDB` contains 2M key-value pairs whose value size is 2KB, and its host queries 8M `Get` requests by randomly generating their keys. This workload characteristic roughly makes `DirectCXL`'s memory accesses be faced with a cache miss for every four queries. Note that `Workload` of `DirectCXL` is longer than that of `KVS`, because `DirectCXL` places all hash table and tree for key-value pairs whereas `KVS` has it in local DRAM. Lastly, all the four graph workloads show similar trends; `Swap` is always slower than `DirectCXL`. They require multiple graph traverses, which frequently generate random memory access patterns. As `Swap` requires exchanging 4KB pages to read 8B pointers for graph traversing, it shows $2.2\times$ worse performance than `DirectCXL`.

5 Conclusion

In this paper, we propose `DIRECTCXL` that connects host processor complex and remote memory resources over CXL's memory protocol (`CXL.mem`). The results of our real system evaluation show that the disaggregated memory resources of `DIRECTCXL` can exhibit DRAM-like performance when the workload can enjoy the host-processor's cache. For real-world applications, it exhibits $3\times$ better performance than `RDMA`-based memory disaggregation, on average.

6 Future Work and Acknowledgement

The authors are extending the kernel for efficient CXL memory management and consider having an SoC silicon as future work of `DirectCXL`. This work is protected by one or more patents. The authors would like to thank the anonymous reviewers for their comments, and Myoungsoo Jung is the corresponding author (mj@camelab.org).

References

- [1] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.
- [2] Ling Liu, Wenqi Cao, Semih Sahin, Qi Zhang, Juhyun Bae, and Yanzhao Wu. Memory disaggregation: Research problems and opportunities. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1664–1673. IEEE, 2019.
- [3] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012.
- [4] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.
- [5] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, 2018.
- [6] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280, 2020.
- [7] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H Peter Hofstee. Thymesisflow: a software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 868–880. IEEE, 2020.
- [8] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 488–504, 2021.
- [9] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiyang Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 417–433, 2022.
- [10] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 79–92, 2021.
- [11] Shin-Yeh Tsai, Yizhou Shan, and Yiyang Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48, 2020.
- [12] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014.
- [13] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.
- [14] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles*, pages 54–70, 2015.
- [15] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, 2015.
- [16] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. Aifm: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332, 2020.
- [17] Gen-Z Consortium. Gen-Z Final Specifications. <https://genzconsortium.org/specifications/>.
- [18] CXL Consortium. Compute Express Link Specification Revision 2.0. <https://www.computeexpresslink.org/download-the-specification>.

- [19] CXL Consortium. Compute Express Link™ 2.0 White Paper. https://www.computeexpresslink.org/_files/ugd/0c1418_14c5283e7f3e40f9b2955c7d0f60bebe.pdf.
- [20] Navin Shenoy. A Milestone in Moving Data. <https://newsroom.intel.com/editorials/milestone-moving-data>.
- [21] Debendra Das Sharma. CXL: Coherency, Memory, and I/O Semantics on PCIe Infrastructure. <https://www.electronicdesign.com/technologies/embedded-revolution/article/21162617/cxl-coherency-memory-and-io-semantics-on-pcie-infrastructure>.
- [22] Patrick Kennedy. Compute Express Link or CXL What it is and Examples. <https://www.servethehome.com/compute-express-link-or-cxl-what-it-is-and-examples/>.
- [23] Hari Subramoni, Ping Lai, Miao Luo, and Dhaleswar K Panda. Rdma over ethernet—a preliminary study. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–9. IEEE, 2009.
- [24] Philip Werner Frey and Gustavo Alonso. Minimizing the hidden cost of rdma. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 553–560. IEEE, 2009.
- [25] Intel. Persistent Memory Developer Kit Version v1.11.0. <https://pmem.io/>.
- [26] Intel. NVDIMM Namespace Specification. https://pmem.io/documents/NVDIMM_Namespace_Spec.pdf.
- [27] UEFI Forum, Inc. Advanced Configuration and Power Interface (ACPI) Specification Version 6.4. <https://uefi.org/specs/ACPI/6.4/>, 2021.
- [28] Linaro. The devicetree specification. <https://www.devicetree.org/>.
- [29] Mellanox. Mellanox ConnectX-3 FDR (56Gbps) Infini-Band VPI. https://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX3_VPI_Card_Dell.pdf.
- [30] Xilinx. Mellanox OpenFabrics Enterprise Distribution. https://www.mellanox.com/products/infiniband-drivers/linux/mlnx_ofed.
- [31] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Malleovich, Iliia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.
- [32] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.
- [33] Alan Bundy and Lincoln Wallen. Breadth-first search. In *Catalogue of artificial intelligence tools*, pages 13–13. Springer, 1984.
- [34] Fan Chung and Linyuan Lu. Connected components in random graphs with given expected degree sequences. *Annals of combinatorics*, 6(2):125–145, 2002.
- [35] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.
- [36] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

Not that Simple: Email Delivery in the 21st Century

Florian Holzbauer
SBA Research

Johanna Ullrich*
University of Vienna*

Martina Lindorfer
TU Wien

Tobias Fiebig
Max-Planck-Institut für Informatik

Abstract

Over the past two decades, the number of RFCs related to email and its security has exploded from below 100 to nearly 500. This embedded the Simple Mail Transfer Protocol (SMTP) into a tree of interdependent and delivery-relevant standards. In this paper, we investigate how far real-world deployments keep up with this increasing complexity of delivery- and security options. To gain an in-depth picture of email delivery apart from the giants in the ecosystem (Gmail, Outlook, etc.), we engage people to send emails to eleven differently configured target domains. Our measurements allow us to evaluate core aspects of email delivery, including security features, DNS configuration, and IP version support on the sending side across different types of providers.

We find that novel technologies are often insufficiently supported, even by large providers. For example, while 65.4% of email providers can resolve hosts via IPv6, only 44.3% can also deliver emails via IPv6. Concerning security features, we observe that less than half (41.5%) of all providers rely on DNSSEC validating resolvers, and encryption is mostly opportunistic, with 89.7% of providers accepting invalid certificates. TLSA, as a DNS-based certificate verification method, is only used by 31.7% of the providers in our study. Finally, we turned our eye to the impact modern standards have on unsolicited bulk email (SPAM). We found that greylisting is effective, reducing the SPAM volume by roughly half while not impacting regular delivery. However, and interestingly, SPAM delivery currently seems to focus on plaintext IPv4 connections, making IPv6-only, TLS-enforcing inbound email servers a more effective anti-SPAM measure—even though it also means rejecting a major portion of legitimate emails.

1 Introduction

Electronic mail (email) relies on the Simple Mail Transfer Protocol (SMTP) for delivery. This protocol was first specified in 1982 in RFC 821 and is now close to celebrating its

*Christian Doppler Laboratory for Security and Quality Improvement in the Production System Lifecycle, Security & Privacy Group, Faculty of Computer Science

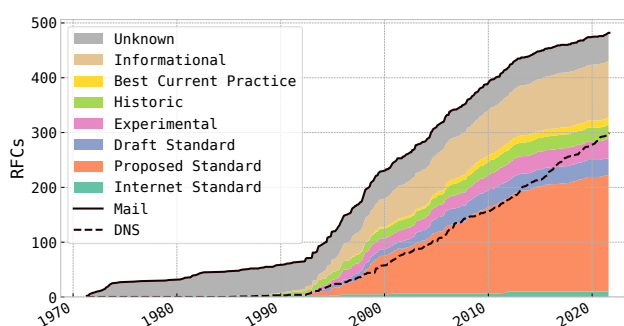


Figure 1: Overview of the explosion of email-related standards (“SMTP Camel”), compared to DNS-related standards.

40th birthday [39]. SMTP had two design goals, namely to allow *reliable* and *efficient* delivery of emails. As with many protocols of the time, security and authenticity were not priorities [16]. In fact, anyone could relay emails through an SMTP server, which was the default configuration for many email servers – like Sendmail – until the late 1990s [3].

However, the practical reality of the Internet led to increased security and authenticity requirements [16]. Since the mid-1990s, hundreds of protocols and extensions have been introduced to cover these gaps, as illustrated in Figure 1. In order to authenticate email, attempts mostly rely on the *Domain Name System* (DNS), which, in turn, suffers from authenticity issues. To address those issues, the *DNS Security Extensions* (DNSSEC) were introduced in 1999, which enabled signing DNS entries [1]. Besides authenticity, the original email protocol faced other security-related challenges, most notably confidentiality, as emails were exchanged in plaintext. In addition to end-to-end encryption approaches like Pretty Good Privacy (PGP) [7], this led to an extension of SMTP for Transport Layer Security (TLS) [18]. Finally, like all protocols on the Internet, SMTP was also affected by the introduction of IPv6.

All these factors have turned the *simple* from SMTP to *complex*. To outline this increase in complexity, we created the *SMTP Camel* in Figure 1 (after the famous DNS Camel of

Bert Hubert, who illustrated the complexity of DNS with “*How many features can we add to this protocol before it breaks?*” [23]). Figure 1 visualizes RFCs related to email – and, for reference, DNS. We compiled this list by performing a title/keyword search on all RFCs on September 28, 2021.¹ In total, we found 481 email-related RFCs compared to 298 DNS-related ones. Among these, more than half of the RFCs belong to the standards track, representing mature standards. We see no development in draft standards as they were declared as deprecated in 2010 [21]. In June 2021, we reached a total of 225 proposed standards. Proposed standards only advance to Internet standards once they have “*widespread deployment of multiple implementations from different code bases*” [21]. Currently, only eleven email-related RFCs have met this requirement, and also the handling of this guideline by the Internet Engineering Task Force (IETF) varies. This indicates that the development of new standards has outpaced their implementation. Furthermore, since the latest email measurement study in 2020 [31], seven new email-related RFCs have been published.

In this paper, we investigate how the increasing number of additional standards has influenced email delivery in the wider ecosystem. Related work already demonstrated that adoption rates of email-related standards are low and implementations often rely on insecure defaults [8, 14, 17, 22, 26, 31, 37, 45]. However, previous work predominantly focused on large operators, such as Google (Gmail) or Microsoft (Outlook), and did not investigate fundamental aspects of email standards, like supported IP versions and the DNS infrastructure of sending systems. We take a step back and investigate the most fundamental aspects of email in transit across a wide sample going beyond major email providers.

To accomplish this, we introduced eleven target address configurations to verify how email providers implement email-related standards and protocols, i.e., we set up systems that – depending on the remote server’s configuration and implementation – either do or do not receive measurement emails. Our measurement technique allows us to measure IP support, STARTTLS configuration, DNSSEC validation, and how different SMTP applications react to greylisting, an anti-SPAM technique by which incoming emails are initially rejected. Our focus is on protocols that influence email delivery once an email has been submitted. To increase the providers’ coverage, we crowdsourced the sending of emails to participants recruited through mailing lists and social media.

As a result, we collect emails from three different sources, spanning (1) small participants in the email ecosystem, (2) large providers, and (3) unsolicited bulk email, aka SPAM. We are the first to discuss the impact of new and established standards on email delivery, as – in contrast to most related measurements – we rely on actively collecting emails, allowing us a more in-depth view of email server configurations.

¹https://www.rfc-editor.org/search/rfc_search_detail.php

In summary, we make the following contributions:

- We introduce a new ranking method using passive data to find the top 15 email providers. Our results highly overlap with Liu et al. [32], while causing significantly less measurement overhead (see Section 3).
- We illustrate challenges in the interoperability between large centralized operators and smaller operators, including how the ability to deliver emails as the main objective limits the adoption of new network and security protocols. We describe how our datasets cover different actors in the email ecosystem in Section 4.
- We are the first to measure and connect the impact of protocol extensions in protocols email relies on – DNS(SEC) and IPv6 – to email delivery and the contrast between smaller and larger providers (see Section 5).
- We illustrate protocol support and compliance in the heavy-tail of the email ecosystem, i.e., in a large set of smaller email operators, and contrast this to earlier work and patterns found in large providers (see Section 6).
- Based on our results, we derive recommendations for email system operators on how they can utilize modern protocol compliance to – currently – reduce SPAM delivery (see Section 7).

Artifacts: Our measurement can be executed using any valid domain and a set of machines connected to the Internet. Along with our paper, we publish a setup-documentation and the scripts we used to receive and analyze emails sent to our systems at <https://github.com/ichdasich/email-measurement-toolchain>. For privacy reasons, we cannot publish our email dataset. This also applies to the SPAM dataset, as even SPAM may contain PII, for example in the recipient addresses.

2 Background: Protocols and Standards

In this paper, we focus on standards influencing email delivery between email servers, i.e., the Mail Transfer Agent (MTA). Email submission, e.g., the communication between Mail User Agent (MUA) and Mail Submission Agent (MSA), is not part of our study. We focus on IP- and DNS-related mechanisms that impact delivery. Interpretations of higher-level delivery security features, like the Sender Policy Framework (SPF) [27], DomainKeys Identified Mail (DKIM) [9], Authenticated Received Chain (ARC) [4], and Domain-based Message Authentication, Reporting, and Conformance (DMARC) [30] are out of scope for our study, as they only influence the receiver’s decision on whether to accept incoming emails or not. We also did not include MTA Strict Transport Security (MTA-STS) in our study as this RFC was too recent

when we set up our infrastructure [33], but Section 3 describes how our work can be extended to include it in the future.

IPv4 [38] and IPv6 [10]. Since addresses in the 2^{32} bit address space of the Internet Protocol Version 4 (IPv4) are running out [41], Internet Protocol Version 6 (IPv6) with a 2^{128} bit address space was introduced in the late 1990s. Two concurrent IP versions introduce a great challenge in terms of interoperability on the network layer, especially as the adoption of IPv6 is still slow [25]. IP version support impacts email delivery *indirectly* via DNS support, i.e., the authoritative and recursive servers support the same IP version, and *directly*, i.e., in terms of whether the involved email servers both support the same IP version. Servers can support IPv4, IPv6, or both—also referred to as “dual-stack.”

DNSSEC [5]. The DNS-Security Extensions (DNSSEC) provide authenticity to DNS responses by signing DNS entries via a keychain along the path of the DNS tree. A DNSSEC validating recursor responds with `SERVFAIL` in case of a validation error. As a consequence, the target domain cannot be resolved, and email delivery fails. Hence, in case of misconfigurations – common in system operations [11] – or attacks, the DNSSEC validation behavior of DNS resolvers at email-sending servers becomes important for email delivery. Similarly, DNSSEC is a prerequisite for DANE (see below).

STARTTLS [19]. The SMTP Service Extension for Secure SMTP over TLS (STARTTLS) enables TLS for email delivery. The connection is established on the same port as SMTP. The original SMTP handshake remains in cleartext. Sending- and receiving servers can (1) not support TLS, (2) support TLS and cleartext, (3) enforce TLS. TLS can be configured either in an (a) opportunistic or (b) strict manner. While opportunistic TLS configurations allow for encrypted connections not validating the remote certificate, strict configurations cause email delivery to fail in case of (1) invalid certificates, (2) not supporting mandatory ciphers, or (3) a connection to a non-TLS-supporting server. In turn, this can then impact email delivery, depending on whether a connection can be established or not.

DANE [20]. The DNS-Based Authentication of Named Entities (DANE) prevents MTA-to-MTA transport encryption from downgrade attacks, even in the absence of certificates signed by a certificate authority (CA); this is done through recording valid CA or end-entity certificates for a domain name via the TLSA DNS record. Trusting/guaranteeing the authenticity of TLSA records (i.e., preventing MITM and DNS cache poisoning scenarios) requires the use of DNSSEC, as described above. Several email server implementations, including Sendmail and Microsoft Exchange, do not yet support requesting TLSA records, in contrast to for example, Postfix and Exim [31].² DANE can be implemented similar to

²Microsoft announced support after our measurement period in Feb, 2022 (see <https://techcommunity.microsoft.com/t5/exchange-team-blog/releasing-outbound-smtp-dane-with-dnssec/ba-p/3100920>)

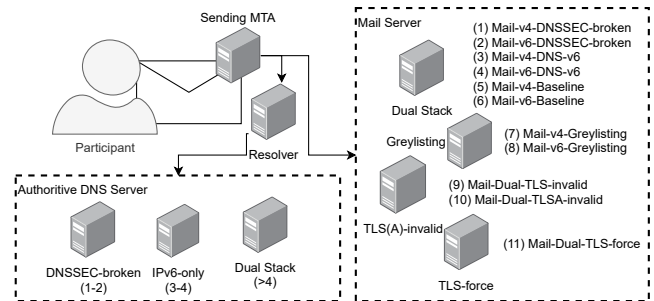


Figure 2: Overview of our measurement setup: 3 DNS servers serve 4 email servers with 11 differently configured target addresses.

TLS in an opportunistic or mandatory manner. Email delivery fails for both opportunistic and mandatory configurations if a signed TLSA record is available but certificate validation fails or for mandatory configurations if no TLSA record can be found.

Anti-SPAM (Greylisting [29]). Greylisting is one of the most simplistic approaches to reduce SPAM emails. It works by initially responding with SMTP code `4xx temporary failure`. While reputable servers usually re-attempt email delivery after several minutes, many SPAM senders do not keep enough state for this. For email delivery, greylisting introduces delays, and email delivery fails if an implementation does not attempt redelivery.

3 Methodology

Measurement Platform. Our measurement setup consists of four email servers running Postfix 3.6 [40] on OpenBSD 6.7 [36] in OpenBSD virtual machines (VMM). As we conduct non-performance bound network measurements, the exact type and model of the used hardware are not relevant to our measurement platform. Furthermore, we rely on three PowerDNS authoritative nameservers in version 4.3.1 to measure the impact of different DNS server setups. We configured a non-default TTL of 300 seconds for all entries in our DNS zones to minimize the impact of caching, i.e., a DNS resolver used by multiple study participants. This also affects our weekly spam domain rotations, pointing them at different measurement target addresses. However, we consider a maximum overlap of five minutes in comparison to a one-week measurement period negligible. IPv6 connectivity to our systems was provided via a Hurricane Electric IPv6 tunnel, while IPv4 connectivity was provided via dedicated IP space from the RIPE region. On these systems, we set up eleven email addresses, as shown in Figure 2. For each of these addresses, we applied different configuration states, which either enable or prevent remote servers from sending emails to them, depending on their own configuration state. This allows us to measure the remote servers’ email delivery capabilities and protocol use by measuring whether they are able to deliver

```
measurement@v4-mail.example.com
measurement@v6-mail.example.com
measurement@v4-mail.v6only.example.com
measurement@v6-mail.v6only.example.com
measurement@v4-mail.dnssec-broken.example.com
measurement@v6-mail.dnssec-broken.example.com
measurement@v4-mail.greylisting.example.com
measurement@v6-mail.greylisting.example.com
measurement@mail-tls-force.example.com
measurement@mail-tls-invalid.example.com
measurement@mail-tlsa-invalid.example.com
```

Figure 3: List of email addresses for the 11 target configurations.

emails to these email addresses. We then asked participants to send *one* email with all measurement addresses in the `To:` field. If we do not receive a message at a specific target address but see in our baseline that the target is included in the `To:` header, we know that the respective feature is not supported. The target addresses can be easily extended to cover new protocols, e.g., MTA-STS [33] was introduced as a barrier against downgrade or interception attacks for domains that are unable to deploy DNSSEC. MTA-STS can be measured by adding two new target addresses in the future. One could implement the TLS-RPT standard to measure TLS reporting frequency, and the other could measure if providers still deliver emails in case of an enforced MTA-STS policy with non-matching MX records.

3.1 Target Address Configurations

We configured the following eleven different email addresses at the unique destination domains listed in Figure 3. Below, we describe the purpose of each of these addresses, i.e., which configuration parameters we tested with them:

IP Support. In order to test basic delivery behavior, we created for both IPv4 (`measurement@v4-mail.`, *Mail-v4-Baseline*) and IPv6 (`measurement@v6-mail.`, *Mail-v6-Baseline*) one address which is configured with no restrictions on delivery. Similarly, we created distinct IPv4- and IPv6 addresses for the DNS and greylisting measurements described below. Note that during our study, we noticed that our choice not to support STARTTLS on this system did indeed introduce an unexpected parameter in the case of senders that enforce STARTTLS use. In turn, this allowed us to detect six providers that enforce STARTTLS for outgoing emails.

DNS Recursion IPv6 Support. To test whether the recursive resolvers of an email sending host support IPv6, we created a subdomain that can only be resolved via IPv6, i.e., the zone had only AAAA glue records, and the hosts in the zone’s NS records also only have AAAA records. Under that domain, we then again created two addresses for IPv4 and IPv6 delivery (`measurement@v4-mail.v6only.`, *Mail-v4-DNS-v6* and `measurement@v6-mail.v6only.`, *Mail-v6-DNS-v6*).

DNSSEC Validation. To test if the remote site validates DNSSEC, we set up a subdomain with a non-matching DS RRset in the parent, i.e., we provide a public key in the parent zone that does not match the key with which records are signed in our zone. Hence, a DNS recursive resolver validating DNSSEC is unable to validate DNSSEC for our domain and should therefore refuse to resolve it. Thus, an email server using a validating resolver cannot deliver emails to that domain. Under that domain, we again created two addresses for IPv4- and IPv6 delivery (`measurement@v4-mail.dnssec-broken.`, *Mail-v4-DNSSEC-broken* and `measurement@v6-mail.dnssec-broken.`, *Mail-v6-DNSSEC-broken*).

TLS Configuration. In order to test the TLS and TLSA behavior of sending hosts, we configured three email addresses that required the use of TLS to deliver emails:

- `measurement@mail-tls-force.`
Mail-Dual-TLS-force on a correctly configured TLS enabled server.
- `measurement@mail-tls-invalid.`
Mail-Dual-TLS-invalid on a server that provides a certificate with a non-matching CN/DNS0 entry.
- `measurement@mail-tlsa-invalid.`
Mail-Dual-TLSA-invalid on a server that has a TLSA record configured, which does not match the supplied certificate.

This setup allows us to verify if systems (1) support STARTTLS, (2) perform opportunistic encryption, and (3) verify TLSA records. Due to a misconfiguration, these systems initially did not support TLS1.3. Hence, remote systems that only support TLS1.3 would be unable to deliver their emails. We were able to isolate the affected cases (76 emails from 29 providers) and reconstructed the actual state from the stored SMTP sessions, as the abort conditions differ between ‘not supporting TLS,’ ‘rejecting the certificate/TLSA record,’ and ‘not having a matching cipher.’

Anti-SPAM (Greylisting). To identify RFC-compliant SMTP implementations, and as an additional control, we set up Postgrey that performs greylisting as an anti-SPAM measure (`measurement@v4-mail-greylisting.`, *Mail-v4-Greylisting* and `measurement@v6-mail-greylisting.`, *Mail-v6-Greylisting*). By configuring these addresses, we can test the impact of greylisting on average SPAM received and check whether legitimate email servers support multiple delivery attempts.

3.2 Email Collection and Recruitment

In order to provide different views on email delivery, we target three types of actors in the email ecosystem: (1) Regular providers by actively engaging users to send emails to our measurement system. (2) A set of top-ranked email providers

Table 1: Recruitment channels for study participants.

Type	Name	Description
Blogs	RIPE Labs APNIC	Article in RIPE's Research Blog/Newsfeed Article in APNIC's Blog/Newsfeed
Social Media	Twitter LinkedIn Reddit	Tweets by researchers involved in the project Posts by researchers involved in the project Reddit post to /selfhosted
Mailing Lists	NANOG INNOG AFNOG SAFNOG DENOG NLNOG IRTF-MAPRG MAIL-OPS	North American Network Operator List Indian Network Operator List African Network Operator List South African Network Operator List German Network Operator List Dutch Network Operator List Network Research Interest Group at IETF/IRTF Global Mail Operator List
Presentations	Internet.nl	Presentation at an organization promoting the adoption of security standards
Personal	-	Colleagues and personal networks, especially in the APNIC and LACNIC regions

by registering user accounts and sending emails. (3) Spammers by registering expired domains and collecting unsolicited emails targeting these domains.

Regular Providers. To collect emails, we actively engaged Internet users to participate in our study. We recruited participants via a social media campaign on Twitter, LinkedIn, and Reddit, via mailing lists focusing on email and network operators, blog articles promoted by Internet governance bodies, and our personal networks (see Table 1). Our recruitment message asked users to visit our website, which provided instructions on how the reader can participate in our study, what the purpose of our study is, and what data access and deletion rights they have. One critical aspect was to ensure that we would be able to distinguish whether an email to one of our measurement hosts was sent and not delivered or not sent at all. Thus, we instructed participants to add all measurement addresses to the `To:` field of a single email. In case a participant's provider performed pre-filtering, e.g., did not accept delivery to domains they cannot resolve, we removed affected emails from the dataset.

Large Providers. In order to rank email providers, we rely on the passively collected Farsight SIE DNS dataset [43]. This enables us to count email servers to which a lot of domains point their MX records, i.e., email servers used for a lot of domains. We assume that the number of domains using a provider's email servers correlates to the provider's size. For our ranking, we use DNSDB MX data extracted for November 2020, which includes data of 73,705,268 different MX lookups. We do not rank providers based on the amount of MX lookups, as low TTLs or different DNS resolver setups might bias the number of lookups. For each MX, we extract the public suffix, i.e., 'example.com' for 'mail.example.com' and 'example.co.uk' for 'mail.example.co.uk' using the Public Suffix List [35]. This results in 23,378,583 different public suffixes. We rank public suffixes of MX records by counting

Table 2: Categories of domains from ExpiredDomains.

Category	Description
1990s	Domains with the first screenshot available on Archive.org between 1990 and 2000 (= "birth year")
alexa	Domains selected based on Alexa traffic rank
backlinks	Domains based on number of Majestic external backlinks
dmoz	Domains found in the latest snapshot of dmoz.org (~2017)
majestic	Domains with low Majestic million global rank
wiki	Domains with high numbers of Wikipedia links

the number of different domains pointing their MX records towards them. We then register accounts at the top 15 providers according to this ranking to send emails to our target domains, as done in prior work [17, 22, 31, 32, 45]. This enables us to compare email delivery from regular providers with an exclusive set of large providers, but also to compare the results of our measurement pipeline to the results of prior work.

Spammers. To collect SPAM emails, we registered expired domains that are still likely to receive SPAM. To do so, we relied on `expirreddomains.net` for a list of domains [42]. To increase the likeliness that respective domains still receive SPAM, we chose them from different categories, based on their age ("birth year," i.e., the first entry in `Archive.org`), their popularity according to rankings from Alexa and Majestic, and the number of links from Wikipedia and the (now defunct) DMOZ content directory. Table 2 lists these categories; Table 3 lists the domains in each category, as well as the volume of SPAM we received during our measurements.

Once registered, we pointed MX records of respective domains at our target domains. To identify if domains still receive SPAM, we executed a three-week baseline measurement. During this period, all 50 re-registered domains pointed their MX records to the MX of *Mail-v4-Baseline*, i.e., our most basic configuration. We classified the domains' value for our measurement based on the amount of SPAM received as *high* (multiple times a week), *low* (once a week), and *none* (none received). To verify that received messages are SPAM, we consulted four active DNS blocklists: `bl.spamcop.net`, `ip.s.backscatterer.org`, `pbl.spamhaus.org` and `sbl.spamhaus.org`. We continuously verified the liveness of these blocklists by requesting IP 127.0.0.2 as a test record.

In total, we found 26% of domains receive SPAM on a regular basis, thus falling into category *high*. In the next step, we pointed high-value SPAM domains towards a set of our target addresses in a weekly rotation until each domain had been pointed at each target at least once. This allowed us to monitor the change in SPAM volume based on the corresponding test conditions. For these measurements, we relied on a reduced set of target addresses. As we only received individual emails and did not simultaneously measure all conditions for each sender, we did not differentiate IPv6 behavior for different target addresses. We only verified general IPv6 support (*Mail-v6-Baseline*), IPv4 sending for IPv6 only DNS (*Mail-*

Table 3: Re-registered domains for SPAM collection and the amount of SPAM emails we received for each of them.

	Category	Domain	Spam Frequency
1	1990s	anx-chicago-rawhide.com	low
2	1990s	intecconstruction.com	high
3	1990s	michael-rauch.com	-
4	1990s	mmf-maintenance.com	high
5	1990s	sapphire-controls.co.uk	high
6	1990s	stratos-bde.com	low
7	alexa	inkpreneur.com	-
8	alexa	jsmmf.org	-
9	alexa	kenyamalikmotors.com	-
10	alexa	lafdo.com	high
11	alexa	nepaltravelcentre.com	high
12	alexa	olakassen.com	-
13	alexa	onmylevelchey.com	-
14	backlinks	18Chaa.com	low
15	backlinks	521qiangweisizu.com	-
16	backlinks	cretms.com	low
17	backlinks	fotits.com	-
18	backlinks	g6china.com	-
19	backlinks	io365f.com	-
20	backlinks	io365i.com	-
21	backlinks	theproxylist.co.uk	-
22	backlinks	tuncayparlak.com	low
23	backlinks	vous-y-etes.com	-
24	dmoz	beechamsdrivingschool.co.uk	high
25	dmoz	bilder-touren-allgaeu.de	-
26	dmoz	costatehogrally.com	low
27	dmoz	djk-handball-coesfeld.de	-
28	dmoz	leben-ohne-alkohol.eu	low
29	dmoz	navesprefabricadassprint.com	high
30	dmoz	parissi.eu	-
31	dmoz	pringfieldfarms.co.uk	-
32	dmoz	printshopleeds.co.uk	low
33	dmoz	smugglegame.com	high
34	dmoz	sotralentz.es	high
35	dmoz	survivalschool.ch	high
36	dmoz	thermoboss.net	low
37	majestic	djmzengaman.com	-
38	majestic	ieican.eu	-
39	majestic	hkmdna.com	-
40	majestic	keerthiwrites.com	-
41	majestic	kientrucnghethuatduongdai.com	-
42	majestic	printspixelz.com	-
43	majestic	studiopaez.com	low
44	majestic	thi-marprojects.be	high
45	wiki	catholic-church-corfu.org	low
46	wiki	grandeguerrafvg.org	-
47	wiki	iranairlinenews.com	-
48	wiki	mosul-network.org	-
49	wiki	unaf-foot.com	-
50	wiki	worldipcomgroup.com	low

v4-DNS-v6), DNSSEC behavior (*Mail-v4-DNSSEC-broken*), as well as our three TLS configurations.

3.3 Ethical Considerations

As our measurements focus on the technical aspects of the involved email setups, this study was not within the scope of our local human subject research ethics council. Nevertheless, we informed participants about the purpose of our data collection, which information we collected, and that they could withdraw from the study at any time. We received one request to be removed from the dataset and complied with this request immediately. In addition, we followed network measurement best practices as outlined in the Menlo report [6, 12].

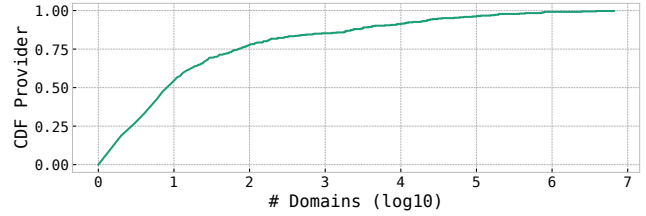


Figure 4: Validation of regular study participants tend to be/use small email providers. We match regular providers to the passive DNS ranking.

This means that we took the necessary technical precautions to protect the only Personally Identifiable Information (PII) we collect, i.e., the sending email addresses. We removed these addresses from our dataset as soon as possible before we started the aggregation of our collected data. Also, since the provider name might reveal PII, we do not publish or share provider names of smaller providers. For our measurements of large providers, we registered accounts ourselves and published their names for better comparison to related work, in accordance with common practice for email-related measurements [17, 22, 31, 32, 45].

4 Datasets

By following our approach, we collected three datasets covering (a) regular providers by volunteers sending emails to our measurement infrastructure, (b) large providers by registering accounts and sending emails ourselves, and (c) spammers by collecting unsolicited emails sent to re-registered domains.

(a) Regular Providers. Between July 4, 2020 and October 29, 2021 we received a total of 5,847 emails. After filtering emails that do not cover all eleven target addresses in the To: field, a total of 4,660 emails sent by 622 study participants remained for further analysis. There is a clear dominance of emails from European countries, see Table 5, a consequence of recruiting via our personal channels (e.g., on Twitter).

Multiple participants used the same infrastructure to send emails; beyond, emails of the same user might be sent by multiple servers in the same domain (e.g. `server1.domain.any` and `server2.domain.any`). Thus, we grouped the data set using the email servers’ first-level domain (EHLO name) at the granularity of providers. This yields a total of 436 providers.

(b) Large Providers. Analysis of the Farsight SIE DNS dataset revealed the top 15 providers as presented in Table 4. We noticed a large gap in served domains even within the top 15 providers, ranging from 14.1% (Google) to 0.68% (1&1) of first-level domains (FLDs) in our passive DNS dataset. The top 15 providers jointly serve 33.8% of all FLDs with MX hosts. To gain an overview of provider sizes in our regular dataset, we matched regular providers with domains in the

Table 4: Top 15 providers based on passive DNS data. Providers greyed out have no online email service, e.g., *Above.com* is a domain broker.

NR	Provider	2015 Durumeric [14]	2015 Foster [17]	2018 Hu [22]	2020 Lee [31]	2021 Tatang [45]	2021 Liu [32]	# Dom.	% Dom.	IP support		DNSSEC		Spam		TLS		
										Mail-v4-Baseline	Mail-v6-Baseline	Mail-v4-DNS-v6	Mail-v6-DNS-v6	Mail-v4-DNSSEC-broken	Mail-v6-DNSSEC-broken	Mail-v4-Greylisting	Mail-v6-Greylisting	Mail-Dual-TLS-force
1	Google	*	△	●	□	◇	○	9,148,093	14.08	✓	✓	✓	✓	✓	✓	✓	✓	✓
2	Microsoft	*			□	◇	○	3,869,507	5.95	✓	✓	✓	✓	✓	✓	✓	✓	✓
3	GoDaddy	*					○	2,453,911	3.78	✓				✓			✓	✓
4	OVHCloud	*					○	1,292,615	1.99	✓				✓			✓	✓
5	Enom						○	871,527	1.34	✓			✓				✓	✓
6	One.com							797,194	1.23	✓				✓			✓	✓
7	Namecheap						○	784,486	1.21	✓		✓		✓			✓	✓
8	Strato						○	762,923	1.17	✓	✓	✓	✓		✓	✓	✓	✓
9	Yandex	*	△				○	759,482	1.17	✓	✓	✓	✓	✓	✓	✓	✓	✓
10	SiteGround						○	712,418	1.10	✓		✓		✓			✓	✓
11	H-email.net							575,451	0.89									
12	Above.com							469,500	0.72									
13	Beget						○	447,284	0.69	✓			✓		✓		✓	✓
14	Tencent	*	△				○	442,064	0.68	✓		✓		✓			✓	✓
15	1&1							440,558	0.68	✓	✓	✓	✓		✓	✓	✓	✓
Optimal Configuration										✓	✓	✓	✓		✓	✓	✓	✓

Table 5: Number of countries/emails/AS per region. Our social media promotion led to an increased number of emails from European countries. We skipped large providers as geographical data has no impact on our provider ranking.

Region	Africa	Asia	Europe	N. America	Oceania	S. America
Regular						
Countries	5	12	30	2	1	3
Emails	48	168	3,368	1,045	1	30
ASes	5	19	202	60	1	3
SPAM						
Countries	22	32	36	15	2	11
Emails	95	2,056	1,963	2,437	17	204
ASes	50	254	234	170	9	119

passive DNS dataset. Figure 4 shows the amount of FLDs pointing at each of the study participants’ domains for email. 80% of regular providers have less than 150 domains relying on them for email service. Comparing our top 15 providers with previous work, we find the largest overlap, namely eleven providers, with Liu et al. [32], who used a five-step approach including MX records, Banner/EHLO messages, and TLS certificates to detect large email providers. Previous work relying on manual ranking results in less overlaps, namely six [14], three [17], two [31, 45], and one [22] (see Table 4), and suggests that human perception of providers is different from their actual dominance in the email ecosystem.

(c) Spammers. We executed SPAM measurements in three phases. First, we conducted a baseline measurement from March 30, 2021 to April 6, 2021. Next, we pointed SPAM domains to our other target addresses in a weekly rotation. Finally, we did another baseline measurement to ensure that the baselines remained stable over our observation time. We received a total of 6,772 unsolicited emails. Thereof, 4,442 (65.7%) were classified as SPAM by one of our four DNS blocklists, suggesting that emails towards the re-registered domains are indeed SPAM. We included all received emails in our further analysis. In comparison to our regular provider dataset, SPAM emails are not dominated by a single region (see Table 5). In comparison to regular and large providers, we can only measure the SPAM volume and its reduction in dependence of the different configurations.

5 Results

For each of the three datasets, namely (a) *regular providers*, (b) *large providers*, and (c) *spammers*, Figure 5 shows the ratio of delivered to undelivered emails per target address. We provide the individual results for the top 15 providers, including a line indicating the optimal configuration, in Table 4. The optimal configuration includes IPv4- and IPv6 support for both email servers and DNS resolvers. Regarding TLS, providers should implement opportunistic STARTTLS, i.e., still use transport encryption when facing self-signed or expired certificates.

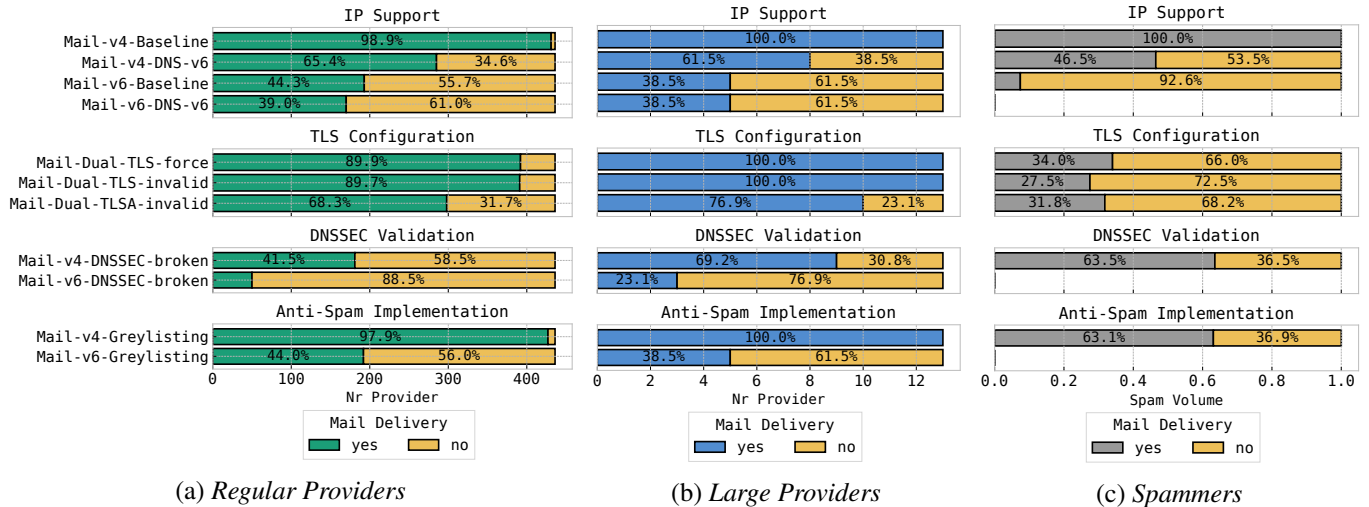


Figure 5: Impact of different target address configurations on email delivery. For our investigation of spammers we skipped the IPv6 target addresses other than the baseline (this affects greylisting, DNSv6, DNSSEC).

However, they should validate TLSA records and reject email delivery in case of an invalid record. As a foundation for DANE and other DNS-based security standards, a provider should rely on a DNSSEC supporting and -validating resolver. Looking at the top 15 providers, we find major discrepancies for even the largest providers. We discuss our measurement results on IP support, TLS configuration, DNSSEC validation, and anti-SPAM implementation in the following sections.

5.1 IP Support

Email Servers. The *Mail-v4-Baseline* is configured without any restrictions on email delivery. For regular providers, however, this baseline is reduced by 5/436 (1.2%) as five providers enforced TLS causing undeliverability (see also Section 3.1). For large providers, the baseline is met by all providers. For spammers, the baseline is necessary to estimate the number of SPAM emails that are typically sent to the investigated domains. For all three populations, the delivery to *Mail-v6-Baseline* is reduced compared to the IPv4 baseline, implying limited deployment of IPv6 at email servers. Differences among regular and large providers remain small – the first received IPv6-only mails in 193/436 (44.3%) of the cases, the latter in 5/13 (38.5%) –, however, SPAM towards the IPv6 target is drastically reduced and accounts for 7.4% of the IPv4 baseline.

DNS Resolvers. Both targets, *Mail-v4-DNS-v6* and *Mail-v6-DNS-v6*, rely on an IPv6-only authoritative nameserver and allow to infer whether resolvers are capable of IPv6. The number of successfully delivered emails to *Mail-v4-DNS-v6* is consistently higher than for IPv6-only email servers (*Mail-v6-Baseline*) – 285/436 (65.4%) vs. 193/436 (44.3%) (regular providers), 8/13 (61.5%) vs. 5/13 (38.5%) (large providers),

Table 6: DNS and email server IP support levels (IPv4 only, IPv6 only or dual stack) of regular providers; reads f.e. 22 (5.0%) have dual stack email servers, but IPv4-only DNS resolver.

DNS	Email			
	IPV4	IPV6	Dual	
IPV4	125	28.7%	1	0.2%
IPV6	0	0.0%	0	0.0%
Dual	116	26.6%	0	0.0%
			22	5.0%
			1	0.2%
			171	39.2%

and 46.5% vs. 7.4% (spammers) – and lead to the conclusion that IPv6 support is more prevalent among DNS resolvers than among email servers. The difference is particularly remarkable for SPAM, and suggests that spammers rely on external DNS resolvers. In comparison to *Mail-v6-Baseline*, delivery towards *Mail-v6-DNS-v6* is, if at all, only slightly reduced – 193/436 (44.3%) vs. 170/436 (39%) (regular providers), and 5/13 (38.5%) vs. 5/13 (38.5%) (large providers) –, i.e., IPv6 support at the email server typically implies IPv6 support at the respective DNS resolver. For the regular providers, Table 6 shows interdependencies concerning IP support: Most dominant are dual stack implementations 171/436 (39.2%) resp. IPv4-only configurations for email and DNS 125/436 (28.7%), as well as IPv4-only email servers with dual stack DNS resolvers 116/436 (26.6%).

Key Findings. In summary, we find that less than half of all regular email providers support IPv6 for their email deployments. Interestingly, IPv6 support for DNS is more frequent, even for providers that do not support IPv6 for their email servers. We conjecture that this is connected to – especially in smaller setups – using public resolvers like the commonly known Cloudflare (1.1.1.1) or Google (8.8.8.8) instances. Interestingly, we also find that 23/436 (5.3%) of the observed

providers *do* use IPv6 for their email setup while *not* using it for their DNS resolvers. Even though finding this case is not unsurprising – PowerDNS, for example, does not perform IPv6 resolution by default—it still means that these operators are not able to deliver emails to IPv6-only zones, even though their email servers support IPv6.

5.2 TLS Configuration

TLS Enforcement. If our target *Mail-Dual-TLS-force* enforces the use of TLS, 392/436 (89.9%) of the regular and all large providers behave accordingly. These numbers indicate a high prevalence of TLS capability among email servers. Concerning SPAM, TLS enforcement has a considerable effect and reduces the number of emails to 34.0%.

TLS Validation. In the presence of invalid certificates, as provided by *Mail-Dual-TLS-invalid*, a similar picture emerges for regular and large providers. As common practice suggests [13] providers regularly fall back on opportunistic STARTTLS. Just one of the regular providers is more strictly configured and rejects email delivery in the case of a certificate with a non-matching CD/DNS0 entry. TLSA mismatch as caused by *Mail-Dual-TLSA-invalid* should technically prevent opportunistic encryption from being used. However, we find that only 138/436 (31.7%) of regular providers and 3/13 (23.1%) of large providers honor the TLSA record and refuse delivery. When we turn our eye to SPAM delivery, we find that enforcing TLS has a significant impact on the number of received emails. On our two TLS-enforcing targets, only 27.5% (*Mail-Dual-TLS-Force*) and 31.8% (*Mail-Dual-TLSA-Invalid*) of the baseline values of emails are received.

Key Findings. The broad majority of providers support TLS. However, emails from 10.1% of regular providers in our dataset would be lost in case of enforcing it. Providers fulfilling TLS enforcement typically also fall back on opportunistic encryption in case of invalid certificates. TLSA – a method to move beyond opportunistic encryption, even in the absence of CA-signed certificates – is sadly ignored by the majority of providers. At the same time, TLS enforcement does not only increase security, but it also reduces SPAM by more than 65%. While spammers could implement TLS quickly, it still would force them to adopt more costly TLS handshakes.

5.3 DNSSEC Validation

Targets *Mail-v4-DNSSEC-broken* and *Mail-v6-DNSSEC-broken* allow to infer the prevalence of resolvers validating DNS records. For regular providers, 181/436 (41.5%) delivered emails to our first target. The remaining 255/436 (58.5%) of all providers conducted a thorough validation for DNSSEC. Among the large providers, DNSSEC validation appears less prevalent: Only 4/13 (30.8%) (IPv4) and 2/5 (40.0%) (IPv6) of providers validate DNSSEC. We suspect that operators

refrain from deploying DNSSEC to avoid customers missing emails or being unable to send emails due to misconfigurations. Furthermore, we observed a significant SPAM reduction for domains with broken DNSSEC. We conjecture that this is due to common open resolvers that validate DNSSEC being regularly used by spammers. This suspicion was confirmed when we revisited our DNS servers' logs to identify the most commonly used DNS resolvers. Query logs are, however, not fully available as log rotations removed some logs due to high response numbers. Still this enabled us to identify the most commonly used DNS resolvers. We were able to match resolvers for 2839/4660 (61%) regular emails and for 3399/6772 (50.2%) of emails sent by spammers. We found 1,443 unique resolver IPs for regular providers and 1,774 for spammers. Relying on MaxMind's public GeoLite AS database, we looked up AS information for each IP. This resulted in 259 unique ASes used for DNS resolution for regular providers and 269 for spammers. Comparing the DNS servers used by regular and large providers with those used by spammers revealed an overlap of 138 IPs and 62 ASes.

Key Findings. DNSSEC validation is performed in 255/436 (58.5%) (IPv4) and 143/193 (74.0%) (IPv6) and regular providers. The numbers for large providers are lower, i.e., 4/13 (30.8%) (IPv4) and 2/5 (40.0%) (IPv6). In comparison, previous work [8] found DNSSEC to be less common; however, those measurements focused on zones using DNSSEC. The numbers for DNSSEC validation among spammers are – surprisingly – comparable to those of large providers. However, this connects to spammers regularly using public resolvers that already validate DNSSEC.

5.4 Anti-SPAM (Greylisting)

The greylisting targets *Mail-v4-Greylisting* and *Mail-v6-Greylisting* provoked an error in delivery the first time and accepted the email in a second – delayed – attempt. Legitimate providers reattempt to deliver emails in case of a failure, and our measurements indeed show that this is the case. Only 4/436 (0.9%) (IPv4) and 1/193 (0.5%) (IPv6) of the regular providers refrain from retransmission, and no large provider does so. However, greylisting reduces the number of received SPAM emails by 36.9%. Interestingly, this makes greylisting a less effective anti-SPAM measure than enforcing TLS.

Key Findings. Greylisting reduces the SPAM volume by 36.9% and does not introduce delivery problems for legitimate email. However, greylisting has less impact than TLS enforcement, which reduces SPAM by over 65%.

6 Related Work

In the past years, email has been receiving significant attention from the research community. In this section, we systematize eleven email-related measurement studies from 2014 onward.

Table 7: Measured adoption rates by related work. Percentages are collected for domains with MX records. SPF, DKIM and DMARC are included for comparison only as they merely influence the receiver’s decision to accept incoming emails.

Citation	Year	Active Meas.	Domains	Sample Size	SPF	DKIM	DMARC	DNSSEC	DANE	TLS (inc.)
Adkins et al. [2]	2014		Facebook	/	-	-	-	-	-	76%
Foster et al. [17]	2015		Alexa	1M	42.3%	-	1%	3.4%	-	-
Foster et al. [17]	2015		Adobe	1M	43.6%	-	0.9%	2.8%	-	54%
Durumeric et al. [14]	2015	•	Gmail	/	-	-	-	-	-	80%
Durumeric et al. [14]	2015		Alexa	1M	47%	-	1.1%	-	-	81.8%
Hu et al. [22]	2018		Alexa	1M	44.9%	-	5.1%	-	-	-
SIDN [44]	2019		.nl	5.9M	44.2%	18.6%	8%	53%	-	62%
Kambourakis et al. [26]	2019/20	•	Custom	3236	80.7%	59.4%	51.3%	23.2%	17.6%	97.6%
Lee et al. [31]	2020		Alexa	100K	-	-	-	-	0.5%	-
Tatang et al. [45]	2021		x	2.04M	50%	13%	11%	-	-	-
Yajima et al. [34]	2021		Tranco	10K	88.7%	-	54.1%	7.7%	0.8%	-
Our work	2020/21	•	Custom	417	91.3%	63%	53.5%	57.4%*	21.6%*	89.9%

*: We can only verify the percentage of DNSSEC resolvers and TLSA validating email servers.
 •: Studies with active measurements
 x: Mix of Alexa top 1M, Tranco, Majestics

We find that these studies use different sample sets and measurement methodologies. Sample sets range from top 1M domain lists to email collections with sample sizes from a million domains to a few thousand. However, using different methodologies, they all ultimately report comparable adoption rates of security-related email protocols, including SPF, DKIM, and DMARC. Hence, we compare their adoption rates and findings to our results in Table 7 to validate our methodology and provide a comprehensive picture of current providers’ email delivery capabilities. Related work on email delivery so far primarily focused on large providers and did not consider the transport perspective – especially IPv6 and DNS – highlighting the gap our work fills.

Adoption Rates. Looking at the reported adoption rates from related work, we do find an upward trend in adoption, especially for security-related standards. We can also observe the difference in adoption rates per region. For example, .nl sees a 53% adoption rate of DNSSEC, which is significantly higher than the, e.g., 7.67% adoption rate for DNSSEC for Tranco Top 10K domains reported by Yajima et al. [34]. We attribute this high adoption rate to the Registrar Scorecard, a campaign incentivizing the deployment of standards by the Dutch domain name registrar SIDN, responsible for the .nl top-level domain [44]. In contrast to the number of DNSSEC-enabled zones, we find the number of validating resolvers to be considerably higher. We find a 57.35% of participants in our study rely on DNSSEC-validating resolvers, mostly due to common public resolvers, for example, the popular 8.8.8.8 resolver offered by Google.

Large Providers. Related work uses several methods for identifying and ranking large email providers (see Table 8): Durumeric et al. [14], Hu et al. [22], and Tang et al. [45] used manual rankings by relying on their own expertise. However, this might induce bias towards the researcher’s experience and location. Foster et al. [17], and Lee et al. [31] relied on email address domains from the leak of Adobe user records

Table 8: Overview of large provider sets used in related work.

Year	Rel. W.	Overlap	Size	Method
2015	Durumeric et al. [14]	6	19	Manually
2015	Foster et al. [17]	3	22	Adobe leak
2018	Hu et al. [22]	1	35	Manually
2020	Lee et al. [31]	2	29	Adobe leak
2021	Tatang et al. [45]	2	25	Manually
2021	Liu et al. [32]	11	15	Custom
2021	Our work		15	passive DNS

in 2013 [28] to rank email providers. However, this approach is limited to a one-time data dump and in completeness as it cannot detect different domains pointing their MX records at the same provider. Liu et al. [32] proposed a more comprehensive approach to detect and rank email providers in 2021. One of their major components is certificate information gathered through Internet-wide SMTP handshakes. In contrast, we introduce a new ranking method based on already existing passive DNS data from DNSDB (see Section 3). Based on this ranking we list the top 15 providers in Table 4. Our method thereby overlaps highly with the results of Liu et al., while introducing significantly less measurement overhead and revealing additional providers.

Sender-side Evaluation. We only found two related measurement studies relevant to the sender-side aspects of email delivery [8, 31]. Chung et al. [8] performed a study focusing on DNSSEC adoption independent of email delivery setups in 2017. They set up ten differently misconfigured target domains (missing, incorrect, expired RRSIGS; missing DNSKEYS; incorrect DS; etc.), collecting data from 4,427 DNSSEC capable resolvers (DO bit set) from the Luminati proxy service. They found that 3,635 (81.1%) failed to validate DNSSEC responses. Only 543 (12.2%) resolvers did handle all ten different scenarios correctly. As we did not focus on DNSSEC validation specifically, but only wanted to test if validation was attempted, we relied on a single DNSSEC

setup for our measurement. Similar to us, Lee et al. [31] used 14 target domains to measure DNSSEC, STARTTLS, and DANE validation in 2020. However, they only measured the top 29 providers ranked by email addresses in the Adobe leak. The measurement setup is similar to ours, but contrary to Lee et al., we actively engaged participants to send emails to our target domains. Hence, we were able to cover a wider range of providers. Our set of large providers also differs from Lee et al. as we used a more comprehensive ranking method, similar to that of Liu et al. [32]. Other studies evaluate email-related protocols from the receiver’s perspective [2, 14, 17, 26], i.e., evaluating emails once they are successfully delivered. For example, studying DNS TXT records between 2015 and 2018, van der Toorn et al. [46] observed a rise in the adoption of email security standards, such as SPF and DKIM, and attributed this to stricter policies from large email providers. However, this line of work generally finds similar problems on the receiver side as we observed on the sender side, e.g., the high complexity of standards, generally low adoption, and therefore, low validation rates. Durumeric [14] found that SPF network ranges are usually configured overly broad, e.g., nearly 30% of domains allow IPv4 address ranges of more than a /16 to originate emails. Furthermore, SPF inclusions are not used carefully, and a multitude of domains trust the same handful of cloud providers. Hu et al. [22] found that 34 of 35 (97%) of popular email providers deliver forged emails to inboxes even if validation of either one or multiples of SPF/DKIM/DMARC failed. Tatang et al. [45] compiled a list of DKIM selectors and found that domains do not only commonly share the same selector, but also the same key.

Standard Complexity. In 2021, Yajima et al. [34] first discussed how standards’ complexity influences their adoption rate. They measured DNS-based security mechanisms and found that setup difficulty influences the adoption rate. Their rating of setup difficulty awards points for the following configuration aspects: DNS record (1pt); DNS server configuration (2pt); email server configuration (2pt); web server configuration (2pt); required third party (3pt). DNSSEC and DANE score the highest with 6 points. While DANE is a relatively new standard introduced in 2012, DNSSEC was introduced in 1999 and still faces a relatively low adoption and validation rate. Potential causes include a (perceived) high risk of service disruptions due to misconfigurations – even in 2021, we still regularly see outages of top-level domains due to misconfigured DNSSEC [24] – and complexity in maintaining DNSSEC. Further investigating the complexity of DNSSEC key material handling, Chung et al. [8] found that a majority of domains roll keys too infrequently, use weak keys, or do not perform rollovers correctly.

7 Discussion

Successful system operation includes design, implementation, and maintenance. In a world of ubiquitous networking, sys-

tems like the email ecosystem cannot be redesigned from scratch, but have to be carefully adapted. This means that successful further development has to consider the impact of improvements on the existing ecosystem. Hence, our measurement provides a perspective on the current state of email.

Our measurements pinpoint an apparent gap between the email ecosystem as standardized by the IETF and its actual deployment. Recently introduced standards such as TLSA (validation) have not made it into practice. Thus, our results suggest that the development of new email standards has to be accompanied by strategies fostering their actual deployment.

7.1 Heavy-tail Email

A pattern that emerges in our measurements as well as in the work of, e.g., Liu et al. [32] is the heavy-tail nature of email: As Table 4 shows, a small portion of operators provide email services to the majority of users and domains on the Internet. Our investigation of related work also shows that studies often focus only on this top part of email providers. However, when we want to understand the email ecosystem, the major challenge is identifying and measuring the diverse tail of email providers and small self-hosted email instances. This becomes particularly challenging if – like in our measurements – user participation is necessary, and might lead to a situation where smaller providers are less investigated with potentially negative impact on their security, resilience, etc.

In a more techno-philosophical dimension, this development also raises concerns in the context of centralization. For example, in 2021 Fiebig et al. measured the migration of universities to large cloud providers, including their email infrastructures [15]. Centralization might accelerate the adoption of standards (e.g., if the relevant players are directly involved in standardization), but this can also potentially enforce the deployment of burdensome standards by small operators, effectively creating a walled garden. Beyond, failure of a single large provider, either due to an accidental error or a deliberate attack, affects a large share of users/domains, emphasizing the importance of decentralization and diversity for the resilience of the overall email ecosystem.

What we certainly highlight – if we want to keep a distributed Internet – is that future development efforts should not only focus on improving standards themselves, but also make it easier to follow these standards and enable operators to run their email infrastructure in full standard compliance. We encourage RFCs drafted by the IETF to be accompanied by *technical and organizational measures facilitating implementation*, reducing the gap between standardization and deployment.

7.2 Delivery vs. Adoption

Looking at the large provider dataset in our study, we find that currently especially large providers prioritize email delivery over security, e.g., DNSSEC validation is enabled for

Google's public DNS service, but not for the resolvers Gmail relies on. This is understandable from an operational standpoint but suggests that security is still considered subordinate to functional goals. We conjecture that Google prioritizes the deliverability of emails over strict enforcement of DNSSEC. The status-quo appears to represent an upside-down world: Precisely for large providers, the deployment of a new security feature appears manageable; yet, they refrain from doing so in a strict manner. At the same time, small operators implement the respective features at a disproportionate operational overhead.

This divergence of the email ecosystem ultimately creates challenges, as new security features often do address actual problems. Hence, the operations community must discuss how this divide can be addressed in the future. The Registrar Scorecard has already proven that financial incentives are successful [44]. Thus, we suggest including the design of such systems already during standardization. The Internet Governance Forum also recommends financial incentives by translation of standards into business cases [47]. However, this poses various challenges, among others the collaboration of multiple stakeholders, funding, and the operation of respective evaluation systems, which have to be solved by future work.

7.3 Standard Deployment and SPAM

In our study, we find that TLS enforcement and IPv6-only delivery have a significant impact on the amount of SPAM systems receive. While IPv6-only delivery naturally has a significant negative impact on legitimate emails being delivered, this impact is smaller when enforcing TLS. According to our measurements, emails from about 10% of regular providers would be affected. However, it is hard to determine an adoption threshold for which enforcement of standards is justified. On the one hand, TLS is an old and well-understood standard, fully supported by large providers which represent the driving force in standard deployment; also the implementation effort is low compared to other standards like DNSSEC or DANE. On the other hand, it is unclear why 10.1% of these providers have not implemented (START)TLS. If this is the case because delivery is still possible without, enforcement of TLS should take place; if the reasons are rooted in structural aspects (e.g., lacking support for certain types of systems or adequately educated staff), we suggest to target these root causes first, again requiring additional technical and organizational measures accompanying RFCs.

8 Conclusion

We investigated email delivery, especially in terms of protocol use (IPv4 vs. IPv6, recursive DNS servers' configuration, TLS sending support) and thereby complement existing related work, which mostly investigated the receiving side of the email ecosystem. Together with a review of related work on

email delivery, this allows us to paint a comprehensive picture of the complexity of email delivery in 2021.

We find that 'new' protocols and extensions relevant to email delivery, like IPv6 and DNSSEC, lack adoption. The overall ecosystem is slow in this regard, especially since large email providers prioritize email delivery and – while trying to offer as many options as possible to receive emails – take a conservative stance when trying to deliver emails to others. This highlights the importance of including the heavy-tail of smaller providers in email-related measurements. Our results show that standard deployment is lower than it could be. At the same time, we know that financial incentives work well to increase deployment rates. Hence, we suggest that such incentive systems should accompany Internet standards. However, continuous funding appears to be difficult; thus, future work should also address the impact of non-financial incentives.

Acknowledgements

This material is based upon work partially supported by (1) the Christian-Doppler-Laboratory for Security and Quality Improvement in the Production System Lifecycle; the financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association are gratefully acknowledged; (2) SBA Research (SBA-K1), a COMET Centre within the framework of COMET – Competence Centers for Excellent Technologies Programme and funded by BMK, BMDW, and the province of Vienna. The COMET Programme is managed by FFG; (3) Project 877110 2big2fail funded by the Program "BRIDGE 1" (FFG); (4) Project FO999887504 DynAISEC funded by the Program "ICT of the Future"—an initiative of the Austrian Ministry of Climate Action, Environment, Energy, Mobility, Innovation and Technology; (5) the European Commission through the H2020 project CyberSecurity4Europe (Grant No. #830929); and (6) by the Vienna Science and Technology Fund (WWTF) through project ICT19-056.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their host institutions or those of the European Commission.

References

- [1] Donald E. Eastlake 3rd. Domain Name System Security Extensions. RFC 2535, RFC Editor, March 1999. <http://www.rfc-editor.org/rfc/rfc2535.txt>.
- [2] M. Adkins. The Current State of SMTP STARTTLS Deployment, 2014. Retrieved Sept. 16, 2021 from <https://www.facebook.com/notes/1453015901605223>.

- [3] Eric Allman. sendmail 8.9.0 released. Retrieved Sept. 20, 2021 from <https://www.sendmail.org/~ca/email/releases/sm890announce.html>.
- [4] Kurt Andersen, Brandon Long, Seth Blank, and Murray Kucherawy. The Authenticated Received Chain (ARC) Protocol. RFC 8617, July 2019. <https://www.rfc-editor.org/info/rfc8617>.
- [5] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033, RFC Editor, March 2005. <http://www.rfc-editor.org/rfc/rfc4033.txt>.
- [6] Michael Bailey, David Dittrich, Erin Kenneally, and Doug Maughan. The Menlo Report. *IEEE Security & Privacy*, 10(2):71–75, 2012.
- [7] J. Callas, L. Donnerhacker, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880, RFC Editor, November 2007. <http://www.rfc-editor.org/rfc/rfc4880.txt>.
- [8] Taejoong Chung, Roland van Rijswijk-Deij, Balakrishnan Chandrasekaran, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, and Christo Wilson. A longitudinal, end-to-end view of the DNSSEC ecosystem. In *Proceedings of the USENIX Security Symposium (USENIX Security 17)*, 2017.
- [9] D. Crocker, T. Hansen, and M. Kucherawy. DomainKeys Identified Mail (DKIM) Signatures. STD 76, RFC Editor, September 2011. <http://www.rfc-editor.org/rfc/rfc6376.txt>.
- [10] Dr. Steve E. Deering and Bob Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 8200, July 2017. <https://rfc-editor.org/rfc/rfc8200.txt>.
- [11] Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. Investigating system operators’ perspective on security misconfigurations. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2018.
- [12] David Dittrich and Erin Kenneally. The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research. Technical report, U.S. Department of Homeland Security, 2012. https://www.dhs.gov/sites/default/files/publications/CSD-MenloPrinciplesCORE-20120803_1.pdf.
- [13] Viktor Dukhovni. Opportunistic Security: Some Protection Most of the Time. RFC 7435, December 2014. <https://rfc-editor.org/rfc/rfc7435.txt>.
- [14] Zakir Durumeric, David Adrian, Ariana Mirian, James Kasten, Elie Bursztein, Nicolas Lidzborski, Kurt Thomas, Vijay Eranti, Michael Bailey, and J Alex Halderman. Neither snow nor rain nor MITM... an empirical analysis of email delivery security. In *Proceedings of the Internet Measurement Conference (IMC)*, 2015.
- [15] Tobias Fiebig, Seda Gürses, Carlos H Gañán, Erna Kotkamp, Fernando Kuipers, Martina Lindorfer, Menghua Prisse, and Taritha Sari. Heads in the clouds: Measuring the implications of universities migrating to public clouds. *arXiv preprint arXiv:2104.09462*, 2021.
- [16] Tobias Fiebig, Franziska Lichtblau, Florian Streibelt, Thorben Krüger, Pieter Lexis, Randy Bush, and Anja Feldmann. Learning from the past: designing secure network protocols. In *Cybersecurity Best Practices*. Springer, 2018.
- [17] Ian D Foster, Jon Larson, Max Masich, Alex C Snoeren, Stefan Savage, and Kirill Levchenko. Security by any other name: On the effectiveness of provider based email security. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.
- [18] P. Hoffman. SMTP Service Extension for Secure SMTP over TLS. RFC 2487, RFC Editor, January 1999. <http://www.rfc-editor.org/rfc/rfc2487.txt>.
- [19] P. Hoffman. SMTP Service Extension for Secure SMTP over Transport Layer Security. RFC 3207, RFC Editor, February 2002. <http://www.rfc-editor.org/rfc/rfc3207.txt>.
- [20] P. Hoffman and J. Schlyter. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698, RFC Editor, August 2012. <http://www.rfc-editor.org/rfc/rfc6698.txt>.
- [21] R. Housley, D. Crocker, and E. Burger. Reducing the Standards Track to Two Maturity Levels. BCP 9, RFC Editor, October 2011. <http://www.rfc-editor.org/rfc/rfc6410.txt>.
- [22] Hang Hu and Gang Wang. End-to-end measurements of email spoofing attacks. In *Proceedings of the USENIX Security Symposium (USENIX Security 18)*, 2018.
- [23] Bert Hubert. DNS-Camel, 2018. Retrieved Jan. 13, 2022 from <https://blog.apnic.net/2018/03/29/the-dns-camel/>.
- [24] IANIX. Major DNSSEC Outages and Validation Failures, November 2021. Retrieved Nov. 16, 2021 from <https://ianix.com/pub/dnssec-outages.html>.

- [25] Siyuan Jia, Matthew Luckie, Bradley Huffaker, Ahmed Elmokashfi, Emile Aben, Kimberly Claffy, and Amogh Dhamdhere. Tracking the deployment of IPv6: Topology, routing and performance. *Computer Networks*, 165:106947, 2019.
- [26] G. Kambourakis, G. Draper, and I. Sanchez. What Email Servers Can Tell to Johnny: An Empirical Study of Provider-to-Provider Email Security. *IEEE Access*, 8:130066–130081, 2020.
- [27] S. Kitterman. Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1. RFC 7208, RFC Editor, April 2014. <http://www.rfc-editor.org/rfc/rfc7208.txt>.
- [28] Brian Krebs. Adobe To Announce Source Code, Customer Data Breach, October 2013. Retrieved Jun. 6, 2022 from <https://krebsonsecurity.com/2013/10/adobe-to-announce-source-code-customer-data-breach/>.
- [29] M. Kucherawy and D. Crocker. Email Greylisting: An Applicability Statement for SMTP. RFC 6647, RFC Editor, June 2012. <http://www.rfc-editor.org/rfc/rfc6647.txt>.
- [30] M. Kucherawy and E. Zwicky. Domain-based Message Authentication, Reporting, and Conformance (DMARC). RFC 7489, RFC Editor, March 2015. <http://www.rfc-editor.org/rfc/rfc7489.txt>.
- [31] Hyeonmin Lee, Aniketh Girish, Roland van Rijswijk-Deij, Taekyoung Kwon, and Taejoong Chung. A Longitudinal and Comprehensive Study of the DANE Ecosystem in Email. In *Proceedings of the USENIX Security Symposium (USENIX Security 20)*, 2020.
- [32] Enze Liu, Gautam Akiwate, Mattijs Jonker, Ariana Mirian, Stefan Savage, and Geoffrey M Voelker. Who’s Got Your Mail? Characterizing Mail Service Provider Usage. In *Proceedings of the ACM Internet Measurement Conference*, 2021.
- [33] D. Margolis, M. Risher, B. Ramakrishnan, A. Brotman, and J. Jones. SMTP MTA Strict Transport Security (MTA-STS). RFC 8461, RFC Editor, September 2018. <http://www.rfc-editor.org/rfc/rfc8461.txt>.
- [34] Yoshiro Yoneya Masanori Yajima, Daiki Chiba and Tatsuya Mori. How prevalent is the operation of DNS security mechanisms? Retrieved Sept. 15, 2021 from <https://indico.dns-oarc.net/event/39/contributions/867/>.
- [35] Mozilla. Public Suffix List, 2021. Retrieved Nov. 24, 2021 from https://publicsuffix.org/list/public_suffix_list.dat.
- [36] OpenBSD. OpenBSD 6.7. Retrieved Oct. 12, 2021 from <https://www.openbsd.org/67.html>.
- [37] Damian Poddebniak, Fabian Ising, Hanno Böck, and Sebastian Schinzel. Why TLS is better without STARTTLS: A Security Analysis of STARTTLS in the Email Context. In *Proceedings of the USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021.
- [38] Jonathan B. Postel. Internet Protocol. RFC 791, September 1981. <https://www.rfc-editor.org/info/rfc791>.
- [39] Jonathan B. Postel. Simple Mail Transfer Protocol. STD 10, RFC Editor, August 1982. <http://www.rfc-editor.org/rfc/rfc821.txt>.
- [40] Postfix. Postfix stable release 3.6.0. Retrieved Oct. 12, 2021 from <http://www.postfix.org/announce/nts/postfix-3.6.0.html>.
- [41] Philipp Richter, Mark Allman, Randy Bush, and Vern Paxson. A primer on IPv4 scarcity. *ACM SIGCOMM Computer Communication Review*, 45(2):21–31, 2015.
- [42] Marco Schmidt. Expired Domains, 2021. Retrieved March 15, 2021 from <https://www.expireddomains.net/>.
- [43] Farsight Security. Passive DNS historical internet database: Farsight DNSDB, 2021. Retrieved Nov. 24, 2021 from <https://www.farsightsecurity.com/solutions/dnsdb/>.
- [44] SIDN. Registrar Scorecard yields great results. Retrieved Sept. 16, 2021 from <https://www.sidn.nl/en/news-and-blogs/registrar-scorecard-yields-great-results>.
- [45] Dennis Tatang, Florian Zettl, and Thorsten Holz. The Evolution of DNS-based Email Authentication: Measuring Adoption and Finding Flaws. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, 2021.
- [46] Olivier van der Toorn, Roland van Rijswijk-Deij, Tobias Fiebig, Martina Lindorfer, and Anna Sperotto. TX-Ting 101: Finding Security Issues in the Long Tail of DNS TXT Records. In *Proceedings of the International Workshop on Traffic Measurements for Cybersecurity (WTMC)*, 2020.
- [47] De Natris Consult Wout de Natris. Setting the Standard for a more Secure and Trustworthy Internet, 2020. Retrieved from https://www.intgovforum.org/multilingual/index.php?q=filedepot_download/9615/2023.

AddrMiner: A Comprehensive Global Active IPv6 Address Discovery System

Guanglei Song^{1,2}, Jiahai Yang^{1,2}, Lin He^{1,2}, Zhiliang Wang^{1,2}, Guo Li¹,
Chenxin Duan¹, Yaozhong Liu^{1,2}, Zhongxiang Sun³

¹*Institute for Network Sciences and Cyberspace, BNRist, Tsinghua University*

²*Quan Cheng Laboratory, Jinan, Shandong, China*

³*School of Computer and Information Technology, Beijing Jiaotong University*

Abstract

Fast Internet-wide scanning is essential for network situational awareness and asset evaluation. However, the vast IPv6 address space makes brute-force scanning infeasible. Although state-of-the-art techniques have made effective attempts, these methods do not work in seedless regions, while the detection efficiency is low in regions with seeds. Moreover, the constructed hitlists with low coverage cannot truly represent the active IPv6 address landscape of the Internet.

This paper introduces *AddrMiner*, a systematic and comprehensive global active IPv6 address probing system. We divide the IPv6 address space regions into three kinds according to the number of seed addresses to discover active IPv6 addresses from scratch, from few to many. For the regions with no seeds, we present *AddrMiner-N*, leveraging an organization association strategy to mine active addresses. It fills the gap of address probing in seedless regions and finds active addresses covering 86.4K IPv6 prefixes announced by BGP, accounting for 81.6% of the probed announced prefixes. For the regions with few seeds, we propose *AddrMiner-F*, utilizing a similarity matching strategy to probe active addresses further. The hit rate of active address probing is improved by 70%-150% compared to existing algorithms. Moreover, for the regions with sufficient seeds, we present *AddrMiner-S* to generate target addresses based on reinforcement learning dynamically. It nearly doubles the hit rate compared to the state-of-the-art algorithms. Finally, we deploy *AddrMiner* and discover 2.1 billion active IPv6 addresses, including 1.7 billion de-aliased active addresses and 0.4 billion aliased addresses, through continuous probing for 13 months. We would like to further open the door of IPv6 measurement studies by publicly releasing *AddrMiner* and sharing our data.

1 Introduction

Internet-wide active address probing is a prerequisite for Internet-scale network surveys. Existing network research and applications rely heavily on Internet-wide active address scanning. For example, the probed active addresses can be used

to examine trends and adoption rates of different technologies [11, 30], measure network topology for reflecting interconnections of nodes [4, 52], probe Internet services for wide-ranging assessments [9, 25] and resource census [23], and test network security by measuring the attack surface [16, 34].

Under IPv4, it is feasible to achieve Internet-wide active address probing by brute-force scanning the entire IPv4 address space at the minute level with high-speed scanning tools such as ZMap [9]. With the rapid development of the Internet as a globally crucial infrastructure, IPv4 no longer meets its development needs, and IPv6 has been promoted and deployed at an accelerated pace worldwide. For example, more than 36.6% of users accessed Google via IPv6 in 2021, compared to fewer than 0.7% in 2012 [21]. However, under IPv6, there are significant challenges to Internet-wide active address discovery. The main reason is that the vast address space of IPv6 makes it more difficult, if not infeasible, to obtain globally active addresses. For example, it would take at least millions of years to scan the entire IPv6 address space using 10 Gigabit links and high-speed scanning tools such as ZMap [9].

To address this issue, researchers usually collect known active IPv6 addresses (i.e., seeds), learn the characteristics of seeds, and generate the target addresses that may have a higher probability of being active for scanning. Although previous research efforts have examined how to detect active IPv6 addresses, the issue of how to perform comprehensive global active IPv6 address discovery remains, mainly in the following aspects:

(1) *Limited usage*. In regions where seeds are missing, existing methods cannot perform effective active IPv6 address probing or even work [7, 15, 18, 24, 33, 36, 47, 51]. This is because they need to learn the characteristics of the seeds to generate target addresses that are more likely to be active. There is still a gap in active IPv6 address probing in regions lacking seeds.

(2) *Limited detection efficiency*. State-of-the-art algorithms [7, 24, 47] have improved the efficiency of active IPv6 address probing. However, these methods are too dependent on seeds. The seed address sampling bias reduces the efficiency of

active address probing because it makes the characteristics of the actual active address inconsistent with those of the seeds.

(3) *Limited coverage*. Previous studies, while building a list of active addresses, called IPv6 hitlist, have tended to be limited to a few IPv6 prefixes announced by BGP [15, 18, 42, 47]. For example, active IPv6 addresses in the latest hitlist [18] cover only 25.5K announced prefixes, which is only $\sim 21.3\%$ of all announced prefixes. They are not truly representative of the active IPv6 address landscape of the Internet. Active detection methods by analyzing seeds are often also limited by the coverage of the seeds [15, 18, 24, 33, 36, 42, 47, 51].

In general, there still lacks a systematic methodology for comprehensive global active IPv6 address probing. To solve the above problems, we design and implement an active address probing system, *AddrMiner* (§4). At its core, *AddrMiner* divides active address probing into three sub-tasks: active IPv6 address probing for 1) address space regions with no seeds, 2) address space regions with few seeds, and 3) address space regions with sufficient seeds, respectively.

First, we present *AddrMiner-N* for the address space regions without seeds (§5). The core idea is based on the observation that address patterns (i.e., structure) tend to have similarities across network configurations. It obtains common patterns by mining the structural features of active addresses collected in other regions, and then migrate to regions without seeds to generate targets for scanning. *AddrMiner-N* leverages graph data structures to describe the similarity of address structure features under different networks (§5.2). Then, it uses graph community discovery algorithms to mine common address structure features for building a common pattern library (§5.3). We observe that the address configuration patterns are more similar within the same network organization than within different organizations. *AddrMiner-N* selects the most similar patterns from the library based on organization association strategy to generate targets (§5.4).

Second, we propose *AddrMiner-F*, an active address probing algorithm for the case where the address space regions contain few seeds (§6). Existing methods cannot effectively learn seed characteristics for active address probing in this scenario due to the lack of seeds. The core idea of *AddrMiner-F* is also to generate targets for probing by selecting the most relevant patterns from the common pattern library and migrating to regions with only a few seeds. Similar to *AddrMiner-N*, *AddrMiner-F* first uses the same method to build a common pattern library (can reuse the one built by *AddrMiner-N*). Then, it improves the efficiency of address detection by extracting relevant patterns from the common pattern library to generate scanning targets using only a few seeds. This is because a few seeds can also provide some information for guiding pattern selection.

Third, we present *AddrMiner-S*, which learns the density characteristic of seeds and corrects density bias to find the real high-density regions of active addresses for address detection,

for the case where the address space regions contain sufficient seeds (§7). The key idea of *AddrMiner-S* is motivated by the higher density regions of active addresses, the higher the hit rate of active addresses. It uses reinforcement learning to update the density distribution of the seeds based on the rewards found for the active addresses and moves toward the actual address distribution to correct the density bias caused by the sampling of seeds.

AddrMiner naturally works in all announced prefix spaces and enables comprehensive active IPv6 address probing in different scenarios by corresponding algorithms to gradually discover active IPv6 addresses from scratch, from few to many.

Contributions. We make the following contributions:

- We present an active IPv6 address probing method, *AddrMiner-N*. It fills the gap of address probing in the seedless address space regions and discovers active IPv6 addresses covering 86.4K prefixes announced by BGP, accounting for 81.6% of all announced prefixes.
- We propose an active IPv6 address probing method, *AddrMiner-F*, which can further discover active IPv6 addresses in address space regions with few seeds. It can find 70%-150% more active addresses than *AddrMiner-N* and the state-of-the-art algorithms.
- We present an efficient active IPv6 address probing method, *AddrMiner-S*, which can efficiently perform active IPv6 address probing in address space regions with sufficient seeds. Compared with state-of-the-art algorithms, the results show *AddrMiner-S* improves the hit rate of active addresses from 28.9% to 56.3%.
- We originally design and implement a global active IPv6 address probing system and discover 2.1 billion active IPv6 addresses, including 1.7 billion de-aliased active addresses and 0.4 billion aliased addresses, through continuous running *AddrMiner* for 13 months. The developed code and continuously probed active addresses are made publicly available at:

<https://github.com/AddrMiner/AddrMiner>

2 Background

In this section, we briefly introduce the background of IPv6 addresses and discuss the characteristics of IPv6 addresses.

IPv6 addresses are 128 bits long. IPv6 unicast addresses consist of a global routing prefix, a local subnet identifier, and an interface identifier (IID). We represent IPv6 addresses in a human-readable text format using eight groups of four hexadecimal characters, each group having 16 bits in total, separated by a colon (“:”). We refer to each hexadecimal character (corresponding to the four bits

of the address) as a nybble. An example IPv6 address is 2001:0db8:0000:0000:0008:8000:200c:417a. To simplify the IPv6 representation, the leading zeros of each group are usually excluded, and the longest all-zero group sequence is replaced with a double colon (“::”). Thus, the simplified representation of the IPv6 address in this example is 2001:db8::8:8000:200c:417a.

IPv6 addresses have the following characteristics. (1) Vastness of IPv6 address space: IPv6 address space is 2^{128} , 2^{96} times of IPv4 address space. This makes active IPv6 addresses very scarce and more hidden, making it a challenging task to find active IPv6 addresses. (2) Diversity of IIDs: IID can be assigned in various ways, such as static configuration [20], stateless address autoconfiguration [37], and DHCPv6 [40]. IPv6 addresses with random IIDs are more difficult to detect.

3 Related Work and Motivation

This section reviews related work and clarifies the motivation of our work on discovering active addresses in the vast IPv6 address space. The existing work can be divided into the following three categories:

Public Resources Extraction. This method obtains active IPv6 addresses through public resource lookup or resolution [6, 13, 18, 50]. DNS is a common and effective channel. Strowes et al. [50] obtained 965K globally routable IPv6 addresses by exhaustively enumerating the reverse DNS domains in the IPv4 address space and performing AAAA queries on the results. Fiebig et al. [13, 14] walked the rDNS tree and collected 5.8M IPv6 addresses. Borgolte et al. [6] also obtained 2.2M IPv6 addresses through DNSSEC-signed reverse zones. Besides, Gasser et al. [18] collected 58.5M IPv6 addresses from public data sources, including Domain Lists [1, 2, 17, 41, 44, 49], FDNS [46], AXFR [35], Bitnodes [54], and RIPE Atlas [38].

Although we can get IPv6 addresses through public resources, in the latest hitlist [18], these addresses only cover 25.5K announced prefixes, which is only $\sim 21.3\%$ of all announced prefixes. Therefore, it is challenging to obtain globally active IPv6 addresses from public sources alone.

Passive Collection. This approach entails passively collecting traffic or log files at vantage points and extracting active IPv6 addresses from them [15, 19, 42, 47]. For the first time, Plonka et al. [42] used IPv6 addresses collected from the activity logs of all customers accessing a global CDN as a dataset and analyzed the characteristics of active IPv6 addresses. Subsequently, Foremski et al. [15] proposed a technique for obtaining potentially active IPv6 addresses from the initial seed dataset. A similar attempt was made by [19, 47] using a large Internet Exchange Point as a vantage point to collect active IPv6 addresses.

However, the above studies have the following shortcomings. First, the vantage point used is not publicly available and is difficult for others to access. Second, to obtain global active

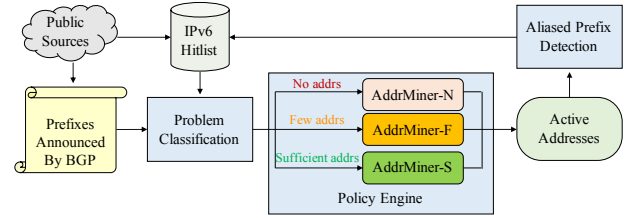


Figure 1: High-level overview of *AddrMiner*

IPv6 addresses, many vantage points need to be deployed worldwide, with high probing overhead. *AddrMiner* removes the vantage point limitation and decreases the threshold for address probing. Any node configured with IPv6 network can use *AddrMiner* to probe active addresses.

Active Address Probing. A viable approach is to discover more active IPv6 addresses by collecting seeds, mining the structural patterns and characteristics of the seeds to generate target addresses, and probing the target addresses [7, 15, 18, 24, 33, 36, 47, 51]. Ullrich et al. [51] proposed a pattern-based recursive algorithm that greedily includes more seeds for scanning each iteration through a variable address range. Entropy/IP [15, 18] learns the internal structural characteristics of seeds to generate target addresses and then scans them to discover active IPv6 addresses. 6Tree [33] and 6Hit [24] combine the hierarchical characteristic of seeds to construct hierarchical space trees, and dynamically guides the direction of address generation based on the probing results. 6Gen [36] and DET [47, 48] use the density characteristic of seeds to detect active IPv6 addresses in high-density regions of seeds. 6GAN [7] aims to discourage aliased address generation via generative adversarial nets with reinforcement learning.

Although all the above methods improve active IPv6 address probing efficiency, the sampling bias of seeds makes the characteristics of actual active addresses inconsistent with those of seeds, resulting in the inability to efficiently generate target addresses for scanning. Although 6Hit attempts to use reinforcement learning to reduce the dependence on seeds, it simply uses the hierarchical characteristic and uses a space repartition mechanism, which leads to random changes in the probe space to reduce the address probing efficiency.

4 Overview of *AddrMiner*

This section describes an active IPv6 address probing system, *AddrMiner*, capable of performing systematic and comprehensive probing of active IPv6 addresses to achieve the accumulation of detected globally active addresses from scratch.

Figure 1 illustrates a high-level overview of *AddrMiner*. It collects seeds to make an IPv6 hitlist from public sources and divides them into different announced prefix spaces. Then, it classifies these announced prefix spaces into different scenar-

ios based on the number of seeds each prefix space contains. The policy engine uses different policies for active address probing according to different scenarios. 1) For announced prefixes with no seeds, *AddrMiner-N* uses the organization association strategy to select candidate patterns to probe for active IPv6 addresses (§5). 2) For announced prefixes with few seeds, *AddrMiner-F* uses the similarity matching strategy to select candidate patterns to discover active addresses further (§6). 3) For announced prefixes with sufficient seeds, *AddrMiner-S* uses reinforcement learning techniques to learn seed address characteristics while circumventing the shortcomings of similar existing schemes to perform active address probing more effectively (§7). The classification of the scenarios for prefix spaces with seeds is discussed in detail in Appendix C. The detected active addresses discovered from the policy engine are tested for aliased prefixes. After eliminating the aliased prefixes, the de-aliased active addresses are the globally active IPv6 addresses and are added to the IPv6 hitlist.

AddrMiner enables comprehensive probing of global active IPv6 address, and provides more and balanced data to support further measurement and security analysis of IPv6 networks.

5 AddrMiner-N

This section presents *AddrMiner-N*, which can guide active address detection under an announced prefix without seeds using patterns of active addresses under other prefixes owned by the same organization to which the prefix without seeds belongs.

5.1 Overview of AddrMiner-N

Since there are no seeds in the address space region, we cannot use seeds to guide active address probes. An effective way to generate targets under such regions is to use specific IPv6 address patterns, i.e., mining the structural characteristics of active addresses collected in other regions and then migrating to regions without seeds to generate targets for scanning. This idea is feasible due to the observation that address patterns tend to have similarities across network configurations [20]. Our analysis of the tens of millions of IPv6 addresses on the Gasser’s hitlist [18] confirms this observation. For example, gateway addresses often have a suffix of ::1 or ::2. Therefore, the crux of the problem is to obtain a common pattern library containing address patterns commonly used in address space regions that have seeds. Our solution, *AddrMiner-N*, is to use undirected graphs to represent the similarity between patterns (§5.2), and then use graph community discovery methods to find communities with high pattern similarity. Each community represents a common address pattern, and these communities construct a common pattern library (§5.3). Finally, it uses the organization association strategy to migrate these

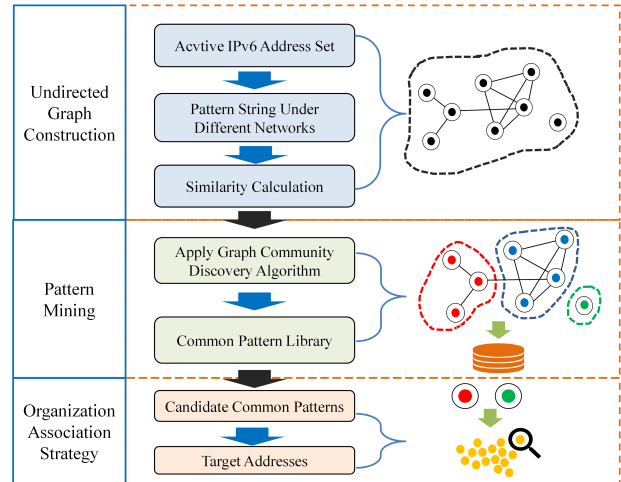


Figure 2: Workflow of *AddrMiner-N*

common address patterns to any announced prefix for address generation (§5.4).

5.2 Undirected Graph Construction

AddrMiner-N constructs an undirected graph to describe the differences between address patterns. The nodes represent the address patterns, and the weights of the edges represent the similarity between different patterns.

Pattern Representation. The address patterns constructed by existing strategies [20, 22, 29, 32, 47] often do not correspond to an accurate probing space. If the space is too large, such as Embedded-IPv4, it will waste a lot of probing resources and reduce the probing efficiency; if the space is too small, such as low-byte, it will limit the probing range and fail to find a large number of active addresses. To solve such a problem, we use **Balanced Spatial Pattern Representation (BSPR)** [31] to extract address patterns. BSPR can accept any IPv6 address set as input and generate flexible patterns representing the structural characteristics of that address set, which can be used to generate targets.

First, the BSPR uses four representations to describe the range of values for any nybble of an IPv6 address, including Single, List, Interval, and Wildcard:

Single: The nybble takes a fixed value, which means that the addresses in the address set do not change in the value taken at that nybble position.

List: The value taken for this nybble is variable, and the range is the set of values taken for the IPv6 addresses in the address set at the nybble position.

Interval: The nybble value is variable. The range is a closed interval consisting of the minimum and maximum values of the IPv6 address set at the nybble position, possibly including values that do not appear at the nybble position.

Wildcard: The nybble value is variable and ranges over

Table 1: The relationship between the four representations and three statistics used by BSPR

ID	Range	Entropy	Values	Representation	Example	Number of values
0	0	0.00	1	Single	a	1
1	$\geq t_r$	$\geq t_e$	$\geq t_c$	Wildcard	*	16
2	$\geq t_r$	$\geq t_e$	$< t_c$	List	[1cf]	3
3	$\geq t_r$	$< t_e$	$\geq t_c$	Interval	[1-e]	14
4	$\geq t_r$	$< t_e$	$< t_c$	List	[2be]	3
5	$< t_r$	$\geq t_e$	$\geq t_c$	Interval	[6-b]	6
6	$\geq t_r$	$\geq t_e$	$< t_c$	List	[679]	3
7	$< t_r$	$< t_e$	$< t_c$	List	[67]	2

all hexadecimal values and may include values that do not appear at the nybble position, as indicated by the wildcard *.

To choose a suitable representation, BSPR introduces three statistics for each nybble of IPv6 address in the input address set: range, Shannon entropy, and value count. The range is equal to the maximum value of the value taken by the nybble minus the minimum value; the Shannon entropy can be calculated according to Formula (5-1); the value count is equal to the number of values that have appeared at the position of the nybble.

$$E(x_i) = - \sum_{v=0x0}^{0xf} p(x_i = v) \log_{16} p(x_i = v) \quad (5-1)$$

The base of Formula (5-1) is 16 because the value count of the nybble is 16, which makes the result of the Shannon entropy calculation fall into the interval [0,1], where x_i represents the i th nybble. The value range of i is [1,32], meaning the 32 nybbles of an IPv6 address. $p(x_i = v)$ can be obtained by dividing the number of IPv6 addresses that take the value v at the i th nybble position in the address set by the total number of addresses.

Table 1 shows how BSPR decides the nybble representation based on these three statistics. Whether these three statistics are large or small is determined by three thresholds t_r , t_e and t_c for range, entropy, and value count respectively.

Second, BSPR needs to solve how to determine the value of the three hyperparameter thresholds. If the thresholds are set too small, the modeled address space increases rapidly. At one extreme, all three hyperparameters are set to 0, and the address generation patterns are all in Wildcard. If the thresholds are set too large, the modeled address space is too small. At the other extreme, all three hyperparameters are taken to their maximum values. The patterns of address generation are all in List, and the value of each nybble represented by List depends entirely on the value of the seed address set, which will aggravate the sample bias.

Suppose the counts of List, Interval, and Wildcard in a pattern are l , r , and w , respectively. L_j , R_j represent the count of values taken at the j th List or Interval, respectively. The value range of Wildcard is the 16 values of a single hexadecimal number. Therefore, the size of the space range (SR) of any

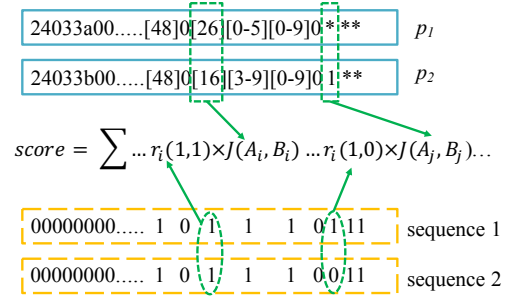


Figure 3: Calculation of the similarity of two patterns

pattern string is thus calculated as follows:

$$SR = 16^w \cdot \prod_{j=1}^l L_j \cdot \prod_{j=1}^r R_j \quad (5-2)$$

Since l , r , and w are affected by the three hyperparameters, t_r , t_e and t_c , SR is a ternary function with respect to these three parameters. The domains of t_r and t_c are the integer of [0,15] and [1,16], respectively. The domain of t_e is the real number of [0,1], which can be discretized, for example, with an interval of 0.05. Let the range of the ternary function SR be the set Y . Since its domain of definition is a finite set, and the range of Y is also a finite set. Let the number of elements of Y be N . The most balanced space range BSR chosen by BSPR should be the average of the Y range. Here, we use the geometric mean because it is less influenced by extreme values than the arithmetic mean.

$$BSR = \sqrt[N]{\prod_{SR_j \in Y} SR_j} \quad (5-3)$$

Third, BSPR can choose the set of hyperparameters when the value of SR is closest to that of BSR as the values of t_r , t_e and t_c , as shown in Formula (5-4). Finally, a pattern is generated based on Table 1.

$$\arg \min_{t_r, t_e, t_c} |SR - BSR| \quad (5-4)$$

Similarity Calculation. The core of constructing undirected edges is to determine between which nodes undirected edges need to be created and the weights of these undirected edges. Since patterns are represented by strings, the common methods of calculating the similarity between strings can also be used. *AddrMiner-N* introduces Jaccard similarity [28] and Hamming distance-based similarity [53] to calculate the similarity between patterns (specific definitions are given in Appendix A).

Figure 3 shows an example of calculating the similarity between two pattern strings. Two main aspects are considered in the calculation: the similarity of the corresponding nybble representation and the similarity of the values taken by the

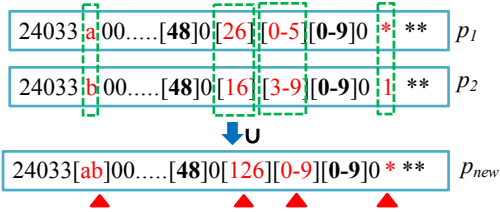


Figure 4: Merging process of different patterns

representation. For the similarity of the corresponding nybble representation, the main focus is to compare whether the two pattern strings have the same fixed value taken at the same nybble position, i.e., whether they belong to the Single representation or not. Non-Single representations include List, Interval, and Wildcard. We convert all Single representation nybbles to a zero value, and Non-Single representations are represented as one, as shown in Sequence 1 and Sequence 2 in Figure 3. In this way, the similarity of the corresponding nybble representation is obtained by calculating the Hamming distance between Sequence 1 and Sequence 2. Regarding the similarity of the values taken by the representation, Jaccard similarity is used to calculate the similarity of the two sets of values of the representation at the same nybble position. Thus, the similarity of the two pattern strings can be obtained by weighting the Jaccard similarity of the corresponding nybble with the Hamming distance-based similarity of each nybble representation as the weight. The calculation is shown in Formula (5-5), where a_i and b_i denote the values of Sequence 1 and Sequence 2 at the i th position, and A_i and B_i indicate the sets of values of pattern strings p_1 and p_2 at the i th nybble representation, respectively.

$$score = \sum_{i=1}^{32} r_i(a_i, b_i) \cdot J(A_i, B_i), \quad (5-5)$$

where r_i indicates Hamming distance-based similarity at the i th nybble and J indicates Jaccard similarity.

Finally, a threshold value h_{min} needs to be determined. If the similarity between two pattern strings exceeds h_{min} , an undirected edge is created. The similarity is used as the weight of that edge. Otherwise, no undirected edge is created. The length of announced prefixes generally does not exceed 56 (14 nybbles). Some prefixes are highly similar, e.g., 2a02:26f0:128:100:/56 and 2a02:26f0:128:500:/56, which causes the merge pattern to contain unannounced prefixes, e.g., 2a02:26f0:128:*00:/56, but addresses in the unannounced space are inactive. Therefore, when constructing the undirected graph, we set h_{min} to 14.0 to avoid generating non-announced spaces as much as possible.

5.3 Pattern Mining

After constructing the undirected graph (§5.2), we apply the community discovery algorithm to cluster similar nodes in

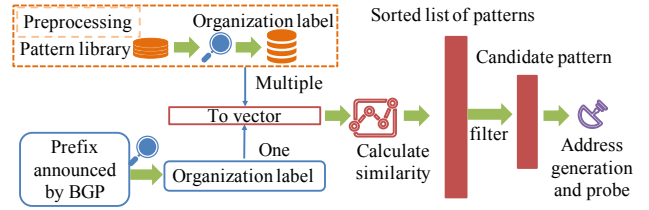


Figure 5: Organization association strategy

undirected graphs and build a common pattern library by mining common patterns from communities.

The graph community discovery algorithm will produce many communities. These nodes have high similarity in the same community but low similarity in different communities. After obtaining the community, we merge the patterns of the nodes contained in the community to extract the common pattern about the community. To make the pattern contain more seeds, we adopt the union method to obtain the pattern. Suppose that C represents a community, which contains k nodes. That is, $C = \{p_1, p_2, \dots, p_k\}$. Then the common pattern of the community is: $p_C = \bigcup_{i=1}^k p_i$. At the nybble positions corresponding to the different patterns, we take a union of the values corresponding to the nybble. Figure 4 shows the merging process of different patterns.

5.4 Organization Association Strategy

To probe active addresses under announced prefixes with no seeds, *AddrMiner-N* adopts an organization association strategy, the core of which is to extract the most relevant patterns of the specified announced prefix from the pattern library and then use them to generate target addresses for scanning. The reason for adopting this strategy is that address configuration patterns are more similar within the same network organization than within different organizations. In the constructed common pattern library, for example, the announced prefixes of organization "Wireless Broadband Service Provider Malaysia" contain the following common patterns: the sixth group of nybbles is represented by Wildcard, and the other nybbles are represented by Single, where the last nybble is 1, and the other nybbles are 0. Specifically, the prefix 2405:7c00:a004::/48 contains the pattern 24057c00a00400000000****00000001 and the prefix 2405:7c00:a000::/48 contains the pattern 24057c00a00000000000****00000001. *AddrMiner-N* makes full use of the organization information of announced prefixes to filter relevant patterns from the pattern library. The evaluation results in §8.1 show that this strategy can enhance significantly improve the probing efficiency.

Figure 5 shows the method of filtering the pattern library using organization labels. We first construct organization labels for the announced prefixes to which the patterns in the

pattern library belong. We obtain the organization label by querying the whois information in Hurricane Electric [10]. To avoid the influence of generic words on the organization association strategy, we remove generic words, such as corporation, international, etc. For example, the organization label of the prefix 2a01:111:2003::/48 is "Microsoft Corporation", we add an English word "Microsoft" to the organization label of the prefix. Similarly, we obtain the organization labels of the announced target prefixes in the same way. To calculate the degree of similarity between the organization labels, we next convert these labels directly into vectors by using the most popular fastText [5, 26, 27] pre-training model in word embedding. Then, we use Euclidean distance to calculate the similarity between the organization label of the target prefix and the organization labels of each pattern in the pattern library. The calculation is shown in Formula (5-6) and yields a list of k_{org} most similar patterns:

$$similarity = \sum_{i \in T, j \in W} d(v_i, v_j), \quad (5-6)$$

where T is the set of words for the organization label of the target prefix, W is the set of words for the organization label of a common pattern in the pattern library. This approach can identify the same network organization, e.g., identifying "Akamai Technologies, inc" and "Akamai International B.V." as belonging to the organization "Akamai" and thus selecting more relevant patterns. Note that when the similarity is small, i.e., the prefixes belong to different organizations, such as "Akamai" and "Fastly", the candidate patterns are randomly selected from the common pattern library.

After obtaining the candidate patterns, we use them to generate target addresses under the target prefix. More specifically, iterate over each candidate pattern to generate a specified number of targets, and then replace the prefixes of the generated target addresses with target prefix. Finally, we probe whether these addresses are active or not.

6 AddrMiner-F

Target regions with few seeds come from both (1) prefix space regions containing few seeds selected from the public IPv6 hitlist, and (2) transformed by detecting few active addresses after running *AddrMiner-N* in regions without seeds. We experimentally find that the active address hit rate of the state-of-the-art algorithm [47] decreases with the number of seeds, especially when the number of seeds is less than 10, the hit rate is already less than 1% (See more details in Appendix C). However, the number of announced prefixes with only few seeds is large. As shown in §8.1, we find more than 30K announced prefixes with less than 10 seeds. To solve this problem, we propose *AddrMiner-F*, which can use few seeds to extract the most relevant patterns from the common pattern library to generate targets for scanning and achieve effective detection of active addresses in announced prefixes

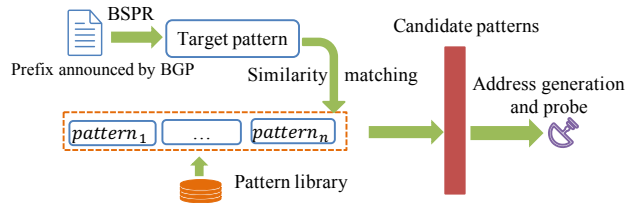


Figure 6: Similarity matching strategy

with few seeds. We call this strategy of matching address patterns using few seeds the similarity matching strategy. *AddrMiner-F* also consists of three steps: undirected graph construction, pattern mining, and similarity matching strategy. Among them, the first two steps have been introduced in §5.2 and §5.3, respectively. The similarity matching strategy is described in detail below.

In this case, we have a small number of IPv6 seeds under the target prefix. Therefore, the similarity matching strategy mainly combines these seeds to filter out a more relevant candidate pattern list from the pattern library. Figure 6 shows the process of similarity matching strategy. We first use BSPR to obtain the patterns of few seeds under the target prefix (i.e., the target pattern). Then we traverse the pattern library and use Formula (5-5) to calculate the similarity between the target pattern and each pattern in the pattern library separately and find the k_{heap} most similar candidate patterns. After getting the candidate patterns, we use the address generation method in §5.4 to generate the target addresses and probe whether they are active or not.

7 AddrMiner-S

Target regions with sufficient seeds come from three scenarios: (1) prefix space regions containing sufficient seeds selected from the public IPv6 hitlist, (2) transformed by detecting sufficient active addresses after running *AddrMiner-N* in regions without seeds, and (3) transformed by detecting sufficient active addresses after running *AddrMiner-F* in regions with few seeds. If the address space regions have enough seeds, the state-of-the-art address generation algorithms based on seeds are effective attempts. They learn the characteristics of seeds to generate target addresses for scanning. However, due to the seeds' sampling bias, the characteristics of the seeds do not coincide with the characteristics of the actual active addresses under the address space regions. The sampling bias reduces the probing efficiency and wastes resources. Although 6Hit attempts to use reinforcement learning to eliminate sampling bias, it simply uses the hierarchical characteristic of seeds and its space repartition mechanism randomly generates target addresses to reduce the efficiency of active address detection. In our work, we propose *AddrMiner-S*, which learns seeds' density characteristic and uses reinforcement learning to correct the discrepancies in the density distribution caused

by the sampling of seeds. In space expansion, *AddrMiner-S* guide the target address generation in a larger address space by merging subspace density characteristic. The model about *AddrMiner-S* is built in Appendix B.

7.1 Target Address Generation Based on Reinforcement Learning

We know the higher the density of active addresses, the higher the hit rate of active addresses (The theoretical proof is in Appendix B). Although the active address density of each region in the real IPv6 network is unknown, we estimate the active address density of each region through Thompson sampling. After discovering seeds' high-density regions, we use the reinforcement learning method to select candidate regions for generating target addresses in the seed address's high-density regions, update the active address density distribution through feedback rewards of each iteration's scanning results, and dynamically adjust the target address generation's direction. As the number of iterations increases, the evaluation of the probability of each action's reward will become more accurate. Eventually, high-density regions of active addresses in the real network will be discovered, and address generation will be performed in the high-density regions.

Space Partition: We first discover the high-density regions $X = \{x_1, x_2, \dots, x_k\}$ of seeds. To quickly cluster the density space distribution of seeds, we use the density space tree [47] to find high-density regions of the seeds in linear time. The root node represents the entire active address space, and the leaf node represents a high-density region of seeds. In each node region x_i , there are two attributes α_i and β_i . Where α_i represents the number of active addresses probed in region x_i , and β_i represents the number of inactive addresses discovered in region x_i . Initially, we take out all the leaf nodes from the density space tree as the high-density regions set X .

After discovering regions with a high density of seeds, we dynamically probe active IPv6 addresses based on reinforcement learning. The iterative process of reinforcement learning consists of three main steps: 1) Generate target addresses to probe (action), 2) Update the reward of probed regions with the number of active addresses and inactive addresses (action's reward) to update the density distribution, and 3) Merge the nodes of the space tree to meet the needs of exploring a larger address space.

Target Generation: To adapt to the large-scale probing of addresses and speed up address probing, we select multiple target regions in each iteration, and the budget (the probing number of target addresses) consumed is b . Since the node region with a larger reward is more likely to discover active addresses, in each iteration, we select the top P searchable regions based on the reward for target address generation in the candidate regions X . We use prior events (action's reward) to evaluate the distribution of active address density. However, the larger space, the higher the risk of searching in the node

region (more difficult to find active addresses). For example, in extreme cases, the hit rate of active addresses is extremely low in the entire IPv6 address space. To reduce the risk of low address probing efficiency due to excessive space, we use the region address variable space (variable dimensions) to adjust the probability of generating active addresses in each region. The number of target addresses generated in each region is calculated as follows:

$$p(x_i) = \frac{e^{R_i}}{\log(V_i) * \sum_{i=1}^n \frac{e^{R_i}}{\log(V_i)}} \quad (7-1)$$

$$N(x_i) = b * p(x_i) \quad (7-2)$$

Where R_i indicates the expected reward in region i , $p(x_i)$ indicates the probability of generating target addresses in region x_i , $N(x_i)$ indicates the number of target addresses generated in region x_i , V_i represents the number of variable dimensions in active addresses in region x_i , and n represents the probing regions of the top P percent of the candidate regions X , b represents the budget consumed per iteration.

Reward Update: We update node regions' reward to increase the chance of generating target addresses in high-density regions for next-round probing. After each round of probing, we need to update the probed node region's reward value based on the probing result. Initially, we take out all the leaf nodes from the density space tree as the high-density regions set X and each leaf node's reward in x_i is initialized as follows:

$$R_i^1 = \text{Beta}(\alpha_i^1, \beta_i^1) \quad (7-3)$$

where R_i represents the expected reward in leaf node region x_i . Initially, α_i^1 is the number of seeds distributed in the leaf node region x_i plus 1, and $\beta_i^1=1$.

After each iteration, the expected reward of the probed region x_i is updated as follows:

$$R_i^{t+1} = \text{Beta}(\alpha_i^t + \alpha^*, \beta_i^t + \beta^*) \quad (7-4)$$

where α^* represents the number of new active addresses from scanning result in node region x_i , and β^* represents the number of new inactive addresses from scanning result in node region x_i . We assume b^* represent the target address generated in the node region x_i in each iteration. α^* , β^* and b^* satisfy the following relationship: $b^* = \alpha^* + \beta^*$.

Node Merging: The search space in the node is defined as the seed address's variable dimensions, but this will cause the search space to be incomplete. We adopt the method of merging upward after the child node's space search is completed, thus ensuring that space not included in the child node can be searched in the parent node.

When a leaf node region needs to be merged, we need to merge all the leaf nodes of the subtree (T) rooted at this leaf node's parent node to ensure that addresses continue to be generated in the high-density region. We can get all

the leaf nodes recursively, but the time consumption is too high. Because the leaf nodes of T are all included in the density regions X to be searched, we can store all the node's child nodes during the tree-building process and only need to intersect with X to get all the leaf nodes when merging. The merging strategy of the node's parameters is as follows:

1) Probed addresses merge: The active addresses (α_f) and inactive addresses (β_f) found in the parent node (f) region is equal to the union of the set of active addresses found in all child nodes ($C = \{x_1, \dots, x_j\}$). The specific relationship is expressed as follows: $\alpha_f = \bigcup_{i=1}^j \alpha_i$ and $\beta_f = \bigcup_{i=1}^j \beta_i$.

2) Reward merge: The parent node's reward value still satisfies beta distribution, and the reward = $Beta(\alpha_f, \beta_f)$ is calculated based on the active and inactive addresses obtained by strategy 1).

3) Space merge: The target address generation space of the parent node is equal to the variable space of the parent node minus the variable space of the child nodes. The specific relationship is expressed as follows:

$$f.var_space = f.var_space - \bigcup_{i=1}^j x_i.var_space.$$

8 Evaluation

This section highlights the evaluation of the effectiveness of active address probing for *AddrMiner*. *AddrMiner* is an active measurement method to discover active addresses. Therefore in our experimental evaluation, we compare *AddrMiner* with active address probing methods, not with passive collection methods (e.g., vantage point mirroring traffic) or public resource extraction methods (e.g., rDNS, Domain Lists, FDNS, AXFR). In all following experiments, we perform aliased prefix detection and aliased address removal.

Data: We automated the process of obtaining Gasser's publicly de-aliased active addresses from December 2020 to June 2021, and obtained 46.2M active IPv6 addresses, covering 49.2K announced prefixes. In addition, we obtained 105,973 announced prefixes from the Pysn project [3]. As shown in Table 2, we classify announced prefix spaces into no seed address spaces, few seed address spaces, and sufficient seed address spaces according to the number of seeds each announced prefix space contains. We have explored the number of seeds on the probing efficiency in Appendix C and selected target regions with the number of seed addresses less than ten as few seed scenarios.

Active Detection: When judging whether the target address is active, we send an ICMPv6 request packet using the ZMap to each address. If we receive a response from an address, we determine that it is active at the time of detection.

Default Parameters: We empirically set the important parameters. In undirected graph construction, h_{min} is set = 14.0. In pattern mining, the Louvain algorithm is used for graph community discovery. Pattern strings with space range SR greater than 10^7 are filtered. In *AddrMiner-S*, we set P to

Table 2: Scenarios classification in the data set

Scenarios Classification	The number of announced prefixes
No seeds	56,730
Few seeds (≤ 10)	31,771
Sufficient seeds	17,472

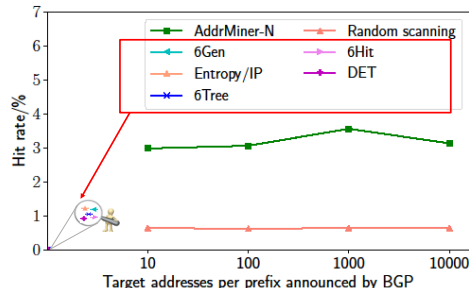


Figure 7: Hit rate of active addresses in the no seed scenario.

0.05, i.e., the top 0.05 percent of the highest reward nodes are selected for probing. We set k_{org} and k_{heap} to 10.0. The maximum number of seeds contained in each leaf node is 4, i.e., $\delta = 4$. We set the granularity of the IPv6 address representation to 4, i.e., $\gamma = 4$.

8.1 Efficiency of *AddrMiner-N*

Suppose b_{count} represents the number of announced prefixes to be probed, p_{count} indicates the number of candidate patterns for each announced prefix, and g denotes the number of addresses generated in each pattern. Thus, the number of addresses generated in each announced prefix is $p_{count} \times g$, and the number of target addresses generated in all announced prefixes is $M = b_{count} \times p_{count} \times g$. We generate different numbers of target addresses for each announced prefix without seeds, i.e., $b_{count} = 56,730$, $p_{count} = 10$, and $g = 1, 10, 100, 1000$.

Figure 7 shows the active address probing results of *AddrMiner-N* and other existing methods in the seedless address scenario. The vertical axis represents the average hit rate of probed prefix spaces without seeds (b_{count}). We found that state-of-the-art target address generation algorithms, including Entropy/IP [15], 6Gen [36], 6Tree [33], 6Hit [24], and DET [47], do not work in seedless regions since they need to learn seeds' characteristics. Compared with random scanning, which has extremely hit rate of only about 0.6%, *AddrMiner-N* has a higher hit rate of up to 3.6% for active address probing.

Furthermore, we perform a more comprehensive probing through the announced prefix space. We use 105,973 announced prefixes as probing regions, employ 500 patterns under each announced prefix, and generate 100 target addresses under each pattern, i.e., $b_{count} = 105,973$, $p_{count} = 500$, and

Table 3: The probing results of the two probing methods

Probing Method	#Active Adrs	#BPFs	Coverage
<i>AddrMiner-N</i>	158,959,500	86,423	81.6%
Random Scanning	708,697	1,421	1.3%

BPFs: prefixes announced by BGP.

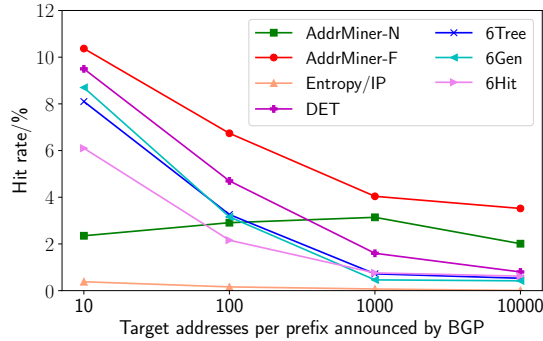


Figure 8: Hit rate of active addresses in the few seed scenario.

$g = 100$. Thus, we generated about 5.2 billion target addresses for probing. Table 3 shows the number of active addresses discovered by the two probing methods and the number of announced prefixes covered by the active addresses. *AddrMiner-N* discovered approximately 159.0M active addresses, covering 86.4K announced prefixes, accounting for 81.6% of all announced prefixes. It indicates that *AddrMiner-N* can perform active address probing over a broader address space and is more suitable for global active IPv6 address probing.

8.2 Efficiency of *AddrMiner-F*

In few seed scenario, we evaluate *AddrMiner-F* by comparing active address hit rate with *AddrMiner-N* and the state-of-the-art target address generation algorithms. We generated different numbers of target addresses among 31,771 announced prefixes containing few seeds, i.e., $b_{count} = 31,771$, $p_{count} = 10$, and $g = 1, 10, 100, 1000$. Figure 8 shows the hit rate of active address probing. When target addresses are small, DET, 6Hit, and 6Tree randomly generate target addresses in low nybble space. As the target addresses increase, the larger the target address will cause the hit rate to decrease. Therefore, in the scenario with few seeds, the state-of-the-art target address generation algorithms are inferior. The hit rate of active addresses is very low when the target space represented by the pattern is too large or too small. We probed all mined common patterns and found that the hit rate was highest when the pattern contained 2 variable nybbles ([32-256] targets). This is because we choose the pattern space closest to the probing number (G) to generate targets randomly. This explains why the hit rate of *AddrMiner-N* increases and then decreases

Table 4: Ratio of common patterns in the pattern library

Patterns	Example of patterns in pattern library	Ratio/%
Low-byte	20010db8000000000000000000000000[1-a]	25.886
Embedded-IPv4	20010db80122034400000000874b2b[3-f][4-f]	7.420
Embedded-port	20010db800000000000000000000[01]***	0.100
ISATAP	fe80000000000000002005efec0000***	0.002
EUI-64	fe80000000000000002aa00ffe3f[2-f][a-c]1c	3.100
Other	240085001000000000de00e300**00**	63.490

in Figure 7 and Figure 8. As the number of target addresses increases, the expansion of the probe space reduces the effectiveness of similarity matching, which affects the efficiency of *AddrMiner-F* probing. However, *AddrMiner-F* is more effective overall than the other methods. When *AddrMiner-F* generates 10-10,000 target addresses for each announced prefix containing few seeds, the active address probing efficiency is improved by 70%-150% compared to existing methods. *AddrMiner-F* enables a more efficient transition from few active address scenario to sufficient address scenario.

8.3 Common Pattern Library Analysis

The common pattern library generated by *AddrMiner* is a set of pattern strings. We analyze this pattern library and provide some further guidance for active IPv6 address probing.

RFC documents [20, 32] presents several common patterns in the IID of IPv6 address, e.g., may be a low-byte IID with a run of zeroes followed only by a low number, an embedded-IPv4 IID inserting one IPv4 address, an embedded-port IID including the service port in the lowest-order byte of the IID, an ISATAP IID with "0200:5EFE" flag and IPv4 address, an EUI-64 IID with an embedded MAC address. Table 4 shows the common address patterns, examples of pattern strings, and the ratio in the pattern library. We find that EUI-64 IID and ISATAP take fixed values at some locations where the address is fixed, so the pattern strings use the Single policy, which is a non-dynamically changing nybble, at these fixed locations. Low-byte address patterns will have more consecutive nybbles in the middle that takes on a value of zero and only change at the end in multiple consecutive nybble positions, such as 20010db800000000000000000000[1-a] for 2001:db8::1 or 2001:db8::4 for these types of IPv6 address structures. The high percentage of the low-byte address pattern in the pattern library means that a brute-force scanning can be attempted for such structures, i.e., fixing the value of the middle nybbles to 0 and traversing only the last consecutive nybbles. It explains the high hit rate of incremental scanning when the number of targets is small. In addition, *AddrMiner* can find many additional address patterns, such as the other types in Table 4, which also account for 63.49% of the total. Although such address patterns do not correspond to address patterns known from RFC documents, their stronger regularity can reduce the difficulty in brute-force scanning of such addresses and improve the efficiency of address probing.

In short, *AddrMiner* can dig out address patterns that not only contain the address patterns of RFC documents, but can also discover more valuable address patterns. We do not consider the assignment of given address space to one of the classes to be static. Furthermore, *AddrMiner* adds newly discovered addresses to the IPv6 hitlist, and updates the common patterns promptly for dynamic IPv6 address spaces.

8.4 Efficiency of *AddrMiner-S*

We evaluate the efficiency of *AddrMiner-S* in probing active addresses by comparing the active address hit rate of *AddrMiner-S* and the state-of-the-art algorithms.

Here, we randomly select announced prefixes that contain more than 1K seeds. We run the above algorithms for each announced prefix to generate target addresses with a budget of 10 times the seeds. Figure 9(a) illustrates the address probing efficiency of *AddrMiner-S* in announced prefixes with sufficient seeds. We observe that the active address hit rate of *AddrMiner-S* outperforms other state-of-the-art algorithms in every announced prefix. In particular, the active address hit rate of *AddrMiner-S* reaches 35.2% in prefix 2001:1291::/32.

To further validate the efficiency of *AddrMiner-S*, we randomly select 1M active addresses as seeds from Gasser’s public hitlist. We use *AddrMiner-S* and state-of-the-art algorithms (Note that 6GAN is not suitable for large-scale global active address detection since the time complexity is too high based on deep learning framework) to generate target addresses with budgets ranging from 10M to 50M. We set the budget b consumed for each iteration to 10K. Figure 9(b) shows the probing results after removing the aliased addresses. We find that *AddrMiner-S* outperforms the other algorithms. When the budget is 50M, the hit rates of the algorithms from highest to lowest are *AddrMiner-S* (56.3%), DET (28.9%), 6Tree (12.9%), 6Gen (14.6%), and Entropy/IP (3.1%), 6Hit (2.6%), and the hit rate of *AddrMiner-S* is almost twice as much. In particular, 6Hit has a high hit rate when the budget is small. Still, space expansion leads to a rapid decrease in hit rate as the budget increases because the target addresses are generated randomly due to the spatial repartition mechanism of 6Hit. *AddrMiner-S* maintains the state learned from sub-space during space expansion to avoid a rapid decrease in hit rate and effectively improve detection.

In the ideal sampling case, the density distribution of seeds is consistent with the density distribution of active addresses in the actual network. The reward (R_i) of each iteration reflects the active address density distribution of the real network. The density distribution of seeds updated by rewards in the next iteration (R_i^{t+1}) is more convergent to the distribution of active addresses in the actual network compared to the previous iteration (R_i^t). Therefore, the similarity between the current seed address density distribution and the actual network’s active address density distribution can be obtained by calculating the difference in reward ranking after each

iteration using Hamming distance. As shown in Figure 9(c), the density distribution of seeds increasingly converges to the density distribution of active addresses in the actual network as the number of iterations increases. In addition, *AddrMiner-S* strikes a balance between exploration and exploitation (the specific analysis is given in Appendix D).

9 IPv6 Hitlist

AddrMiner probes each IPv6 prefix announced by BGP, requiring approximately one month to probe all announced prefixes. As the number of all announced prefixes exceeds 100K, this results in a long probing time. Therefore, to deal with the dynamic changes in the IPv6 space, we repeat the probing of IPv6 prefixes announced by BGP every month. The probe period should be set as short as possible to get a more accurate view of active IPv6 addresses, depending on the probe resources. We have developed *AddrMiner* for continuously probing active IPv6 addresses worldwide for 13 months and discovered 2.1B active addresses (covering 86.4K announced prefixes), including 1.7 billion de-aliased active addresses (IPv6 hitlist) and 0.4 billion aliased addresses. Meanwhile, we found 1.1M aliased prefixes, which are described and analyzed in Appendix E. The IPv6 hitlist is analyzed as follows:

Time Characteristics. Active IPv6 addresses have time characteristics. IPv6 addresses, especially client addresses, have a short lifetime. Therefore, when a probe response is received, we can only determine that the IPv6 address is active at the response time. We analyze the stability of addresses to determine addresses’ lifetime, mainly by separating server addresses, router addresses, persistent or stable client addresses, and temporarily active client addresses.

We define nd -stable to represent the stability of the address. For example, 1d-stable is active for at least one day during the continuous detection period, and nd -stable address means active for at least n days. The active address of nd -stable is also the address of $(n-1)d$ -stable. We send an ICMPv6 request to each address we collect every day and record these addresses’ lifetime according to the response information from January 8, 2021. As shown in Table 5, we found that the long-term active addresses(100d-stable addresses) are more than 46%. Compared with temporarily active client addresses, long-term active addresses are more meaningful for detection.

IID Analysis. We analyze the IID types of active address assignments to understand the global IPv6 address configuration landscape. Utilizing the `addr6` tool [12], we have divided the IID portion of IPv6 addresses into different types.

In Table 5, we analyze the IID allocation types of different stable addresses. The 1.7 billion 1-stable de-aliased addresses mean IPv6 hitlist we collected. We found that pattern-bytes (some discernible patterns) IID addresses accounted for as high as 40.8%, closely related to our detection strategy because we mainly detect active addresses by constructing common pattern library (similarly, low-byte IID and embedded-

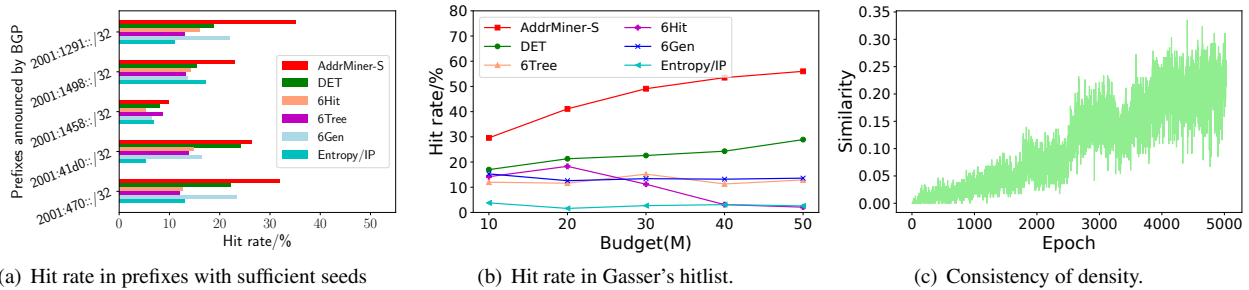


Figure 9: Comparisons between the *AddrMiner-S* and the state-of-the-art algorithms with sufficient seeds.

Table 5: IID Analysis of Discovered n-stable Addresses

-	#IPs	EUI-64	Embedded-IPv4	Pattern-bytes	Randomized	Low-byte
1d-stable(Hitlist)	1.7B	71.4M (4.2%)	251.6M (14.8%)	676.6M (39.8%)	411.4M (24.2%)	277.1M (16.3%)
7d-stable	1.1B (65.8%)	57.8M (3.4%)	212.5M (12.5%)	506.6M (29.8%)	113.9M (6.7%)	227.8M (13.4%)
30d-stable	919.4M (54.1%)	760.8K (0.0%)	204.0M (12.0%)	498.1M (29.3%)	13.6M (0.8%)	202.3M (11.9%)
60d-stable	860.2M (50.6%)	701.6K (0.0%)	190.4M (11.2%)	464.1M (27.3%)	13.5M (0.8%)	188.7M (11.1%)
100d-stable	783.7M (46.1%)	680.4K (0.0%)	173.4M (10.2%)	425.0M (25.0%)	10.3M (0.6%)	173.3M (10.2%)

IPv4 IID addresses). In addition, the IPv6 hitlist contains 24.2% of randomized IID addresses, which are randomly generated in high-density regions. The EUI-64 IID addresses are only 4.2%. This is because Gasser’s hitlist contains a small proportion of EUI-64 IID addresses. At the same time, the detected address space is small, and there is no address generation in the EUI-64 flag. In vertical analysis, we found that randomized IID and EUI-64 IID addresses are more unstable during continuous detection. The proportion of temporarily active client addresses is high, and the lifetime is less than seven days. Embedded-IPv4 IID, low-byte IID, and pattern-bytes IID addresses have high stability and a long lifetime. These are more likely to contain long-term active and stable client addresses, server addresses, router addresses, etc.

We further analyze the organization and location distribution of active addresses in the IPv6 hitlist in Appendix F.

10 Ethical Considerations

To perform global IPv6 address probing, we follow ethical conventions for network measurement, including recommendations provided by Partridge et al. [39] and Dittrich et al. [8]. We first evaluate whether active address measurements induce harm to the probed hosts and networks. We send only one probe packet to each IP address, which minimally affects the host and the network where the IP is located. To avoid duplicate probes, *AddrMiner* removes IPv6 addresses that have already been probed from the generated target addresses. Next, we evaluate whether the probing behavior will cause harm to the local network. We will use distributed probes with a probing rate limit of 10 Mbps per probe to avoid causing problems to the network where the probing point is located during active address probing.

11 Conclusion and Future Work

This work proposes a systematic methodology, *AddrMiner*, which comprehensively probes the global active IPv6 addresses. We follow ethical conventions for network measurement. *AddrMiner* divides the global active IPv6 address probing into three scenarios and accumulates active addresses from none to many. We used *AddrMiner* to probe the global active IPv6 addresses and found 2.1 billion active addresses within 13 months. *AddrMiner* removes the limitation of using a vantage point for active IPv6 address probing. Our work will effectively support more researchers to conduct in-depth IPv6 network measurement and security research. We share code and data at: <https://github.com/AddrMiner/AddrMiner>.

In future work, we will continue to probe active IPv6 addresses. In addition, the blocking strategies and middle boxes can affect the detection of active addresses [25], we will further study their impact on active address detection.

12 Acknowledgments

We would like to thank our shepherd, Adrian Perrig, and the anonymous reviewers for their insightful comments. We also thank Chenglong Li, Enhuan Dong, Yichao Wu, Jinjin Wei, Jinlei Lin, Long Pan, Hao Gao, Yirui Luo, and Leyao Nie for their feedback and suggestions. This work is supported by the National Key Research and Development Program of China under Grant No. 2018YFB1800200 and Beijing Natural Science Foundation under Grant No.4222026. Lin He and Jiahai Yang are the corresponding authors of this paper.

References

- [1] Johanna Amann, Oliver Gasser, Quirin Scheitle, Lexi Brent, Georg Carle, and Ralph Holz. Mission accomplished?: HTTPS security after diginotar. In *Proceedings of the 2017 Internet Measurement Conference*, pages 325–340. ACM, 2017.
- [2] APWG. Apwg: Cross-industry global group supporting tackling the phishing menace. <http://antiphishing.org>, 2018.
- [3] Hadi Asghari and Arman Noroozian. Pyasn. <https://pypi.org/project/pyasn/>, 2020.
- [4] Robert Beverly, Ramakrishnan Durairajan, David Plonka, and Justin P Rohrer. In the ip of the beholder: Strategies for active ipv6 topology discovery. In *Proceedings of the 2018 Internet Measurement Conference*, pages 308–321, 2018.
- [5] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomáš Mikolov. Enriching word vectors with subword information. *Trans. Assoc. Comput. Linguistics*, 5:135–146, 2017.
- [6] Kevin Borgolte, Shuang Hao, Tobias Fiebig, and Giovanni Vigna. Enumerating active ipv6 hosts for large-scale security scans via dnssec-signed reverse zones. In *2018 IEEE Symposium on Security and Privacy (S&P)*, pages 770–784, 2018.
- [7] Tianyu Cui, Gaopeng Gou, Gang Xiong, Chang Liu, Peipei Fu, and Zhen Li. 6gan: Ipv6 multi-pattern target generation via generative adversarial nets with reinforcement learning. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021.
- [8] David Dittrich, Erin Kenneally, et al. The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research. Technical report, US Department of Homeland Security, 2012.
- [9] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 605–620, 2013.
- [10] Hurricane Electric. Hurricane electric bgp toolkit. <https://bgp.he.net/>, 2021.
- [11] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. Measuring https adoption on the web. In *26th USENIX security symposium (USENIX security 17)*, pages 1323–1338, 2017.
- [12] F.Gont. Ipv6 toolkit. <https://www.sixnetworks.com/research/tools/ipv6toolkit>, 2021.
- [13] Tobias Fiebig, Kevin Borgolte, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Something from nothing (there): collecting global ipv6 datasets from dns. In *International Conference on Passive and Active Network Measurement*, pages 30–43. Springer, 2017.
- [14] Tobias Fiebig, Kevin Borgolte, Shuang Hao, Christopher Kruegel, Giovanni Vigna, and Anja Feldmann. In rdns we trust: revisiting a common data-source’s reliability. In *International Conference on Passive and Active Network Measurement*, pages 131–145. Springer, 2018.
- [15] Pawel Foremski, David Plonka, and Arthur Berger. Entropy/ip: Uncovering structure in ipv6 addresses. In *Proceedings of the 2016 Internet Measurement Conference*, pages 167–181, 2016.
- [16] Kensuke Fukuda and John Heidemann. Who knocks at the ipv6 door? detecting ipv6 scanning. In *Proceedings of the Internet Measurement Conference 2018*, pages 231–237, 2018.
- [17] Oliver Gasser, Benjamin Hof, Max Helm, Maciej Korczynski, Ralph Holz, and Georg Carle. In log we trust: Revealing poor security practices with certificate transparency logs and internet measurements. In *PAM*, volume 10771 of *Lecture Notes in Computer Science*, pages 173–185. Springer, 2018.
- [18] Oliver Gasser, Quirin Scheitle, Pawel Foremski, Qasim Lone, Maciej Korczyński, Stephen D Strowes, Luuk Hendriks, and Georg Carle. Clusters in the expanse: Understanding and unbiasing ipv6 hitlists. In *Proceedings of the 2018 Internet Measurement Conference*, pages 364–378, 2018.
- [19] Oliver Gasser, Quirin Scheitle, Sebastian Gebhard, and Georg Carle. Scanning the ipv6 internet: towards a comprehensive hitlist. *arXiv preprint arXiv:1607.05179*, 2016.
- [20] Fernando Gont and Tim Chown. Network Reconnaissance in IPv6 Networks. RFC 7707, March 2016.
- [21] Google. Google ipv6. <https://www.google.com/intl/en/ipv6/statistics.html>, 2021.
- [22] Lin He, Gang Ren, Ying Liu, and Jiahai Yang. Pavi: Bootstrapping accountability and privacy to ipv6 internet. *IEEE/ACM Transactions on Networking*, 29(2):695–708, 2021.
- [23] John Heidemann, Yuri Pradkin, Ramesh Govindan, Christos Papadopoulos, Genevieve Bartlett, and Joseph Bannister. Census and survey of the visible internet. In

Proceedings of the 8th ACM SIGCOMM conference on Internet measurement, pages 169–182, 2008.

- [24] Bingnan Hou, Zhiping Cai, Kui Wu, Jinshu Su, and Yinqiao Xiong. 6Hit: A reinforcement learning-based approach to target generation for internet-wide ipv6 scanning. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021.
- [25] Liz Izhikevich, Renata Teixeira, and Zakir Durumeric. LZR: Identifying unexpected internet services. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [26] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hervé Jégou, and Tomáš Mikolov. Fast-text.zip: Compressing text classification models. *CoRR*, abs/1612.03651, 2016.
- [27] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomáš Mikolov. Bag of tricks for efficient text classification. In *EACL (2)*, pages 427–431. Association for Computational Linguistics, 2017.
- [28] Fatih Karabiber. Jaccard similarity. <https://www.learnatasoci.com/glossary/jaccard-similarity/>.
- [29] Seiichi Kawamura and Masanobu Kawashima. A Recommendation for IPv6 Address Text Representation. RFC 5952, August 2010.
- [30] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. Coming of age: A longitudinal study of tls deployment. In *Proceedings of the 2018 Internet Measurement Conference*, pages 415–428, 2018.
- [31] Guo Li, Lin He, Guanglei Song, Zhiliang Wang, Jiahai Yang, Jinlei Lin, and Hao Gao. Ipv6 active address discovery algorithm based on multi-level classification and space modeling. *Journal of Tsinghua University (Science and Technology)*, 61(10):1177–1185, 2021.
- [32] Xing Li, Mohamed Boucadair, Christian Huitema, Marcelo Bagnulo, and Congxiao Bao. IPv6 Addressing of IPv4/IPv6 Translators. RFC 6052, October 2010.
- [33] Zhizhu Liu, Yinqiao Xiong, Xin Liu, Wei Xie, and Peidong Zhu. 6tree: Efficient dynamic discovery of active addresses in the ipv6 address space. *Computer Networks*, 155:31–46, 2019.
- [34] Soo-Jin Moon, Yucheng Yin, Rahul Anand Sharma, Yifei Yuan, Jonathan M Spring, and Vyas Sekar. Accurately measuring global risk of amplification attacks using ampmap. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [35] Ayman Mukaddam, Imad H. Elhajj, Ayman I. Kayssi, and Ali Chehab. IP spoofing detection using modified hop count. In *AINA*, pages 512–516. IEEE Computer Society, 2014.
- [36] Austin Murdock, Frank Li, Paul Bramsen, Zakir Durumeric, and Vern Paxson. Target generation for internet-wide ipv6 scanning. In *Proceedings of the 2017 Internet Measurement Conference*, pages 242–253, 2017.
- [37] Dr. Thomas Narten, Tatsuya Jinmei, and Dr. Susan Thomson. IPv6 Stateless Address Autoconfiguration. RFC 4862, September 2007.
- [38] RIPE NCC. Ipmap. <https://ftp.ripe.net/ripe/ipmap/>, 2018.
- [39] Craig Partridge and Mark Allman. Ethical Considerations in Network Measurement Papers. *Communications of the ACM*, 59(10):58–64, 2016.
- [40] Charles E. Perkins, Bernie Volz, Ted Lemon, Michael Carney, and Jim Bound. Dynamic Host Configuration Protocol for IPv6 (DHCPv6). RFC 3315, July 2003.
- [41] PhishTank. A nonprofit anti-phishing organization. <http://www.phishtank.com>, 2018.
- [42] David Plonka and Arthur Berger. Temporal and spatial classification of active ipv6 addresses. In *Proceedings of the 2015 Internet Measurement Conference*, pages 509–522, 2015.
- [43] Daniel J. Russo, Benjamin Van Roy, and Abbas Kazerouni. A tutorial on thompson sampling. <https://www.overleaf.com/project/60827af280c8e85011d3b800>, 2021.
- [44] Quirin Scheitle, Taejoong Chung, Jens Hiller, Oliver Gasser, Johannes Naab, Roland van Rijswijk-Deij, Oliver Hohlfeld, Ralph Holz, David R. Choffnes, Alan Mislove, and Georg Carle. A first look at certification authority authorization (CAA). *Comput. Commun. Rev.*, 48(2):10–23, 2018.
- [45] Aleksandrs Slivkins. Introduction to multi-armed bandits. <https://arxiv.org/pdf/1904.07272.pdf>, 2019.
- [46] Rapid7 Project Sonar. Forward dns data. https://opendata.rapid7.com/sonar.fdns_v2/, 2018.
- [47] Guanglei Song, Lin He, Zhiliang Wang, Jiahai Yang, Tao Jin, Jiuling Liu, and Guo Li. Towards the construction of global ipv6 hitlist and efficient probing of ipv6 address space. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2020.

- [48] Guanglei Song, Jiahai Yang, Zhiliang Wang, Lin He, Jinlei Lin, Long Pan, Chenxin Duan, and Xiaowen Quan. Det: Enabling efficient probing of ipv6 active addresses. *IEEE/ACM Transactions on Networking*, 2022.
- [49] Spamhaus. The spamhaus project6. <https://www.spamhaus.org>, 2018.
- [50] Stephen D Strowes. Bootstrapping active ipv6 measurement with ipv4 and public dns. *arXiv preprint arXiv:1710.08536*, 2017.
- [51] Johanna Ullrich, Peter Kieseberg, Katharina Krombholz, and Edgar Weippl. On reconnaissance with ipv6: a pattern-based scanning approach. In *2015 10th International Conference on Availability, Reliability and Security*, pages 186–192. IEEE, 2015.
- [52] Kevin Vermeulen, Justin P Rohrer, Robert Beverly, Olivier Fourmaux, and Timur Friedman. Diamondminer: Comprehensive discovery of the internet’s topology diamonds. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 479–493, 2020.
- [53] Wikipedia. Hamming distance. <https://www.tutorialspoint.com/what-is-hamming-distance>, 2022.
- [54] Addy Yeowr. Bitnodes api. <https://bitnodes.earn.com/>, 2018.

A Similarity Definition

In this section, we give the definition of Jaccard similarity and Hamming distance-based similarity.

- **Jaccard similarity:** Jaccard similarity can be used to calculate the similarity between any two sets. The calculation is shown in Formula (1), where U_1 and U_2 are both sets. In particular, if U_1 and U_2 are both empty sets, then $J(U_1, U_2)$ is 0.

$$J(U_1, U_2) = \frac{|U_1 \cap U_2|}{|U_1 \cup U_2|} \quad (1)$$

- **Hamming distance-based similarity:** For two sequences of the same length z_1 and z_2 , their similarity based on Hamming distance is calculated as follows:

$$S_{HD} = \sum_{i=1}^n r(z_1[i], z_2[i])$$

$$r(a, b) = \begin{cases} 1, & \text{if } a = b \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

B Model Building of AddrMiner-S

We use a multi-armed bandit model [45] based on Thompson sampling [43] to dynamically update the density distribution in the active address space to correct the inconsistency in the density distribution between the seeds and the actual active addresses.

We divide the IPv6 address space into different density regions $X = \{x_1, x_2, \dots, x_k\}$, and each address region is an arm of the multi-armed bandits. There are k actions $A = \{a_1, a_2, \dots, a_k\}$, and a_i refers to scanning the target address in x_i and probing whether it is an active address, where $i \in [1, k]$. $\Theta = \{\theta_1, \theta_2, \dots, \theta_k\}$ represents the mean reward. The distribution of each arm reward is the Bernoulli score with Θ as the parameter:

$$P(r|a_i, \theta_i) = \begin{cases} \theta_i, & \text{if } r = 1 \\ 1 - \theta_i, & \text{otherwise} \end{cases} \quad (3)$$

When a_i is played, the action produces a reward of one with probability θ_i , and a reward of zero with probability $1 - \theta_i$. The θ_i can be interpreted as an action’s success probability or mean reward. Let the agent begin with an independent prior belief over each θ_i . Take these priors to be beta-distributed with parameters $A = \{\alpha_1, \dots, \alpha_k\}$ and $B = \{\beta_1, \dots, \beta_k\}$. In particular, for each action a_i , the prior probability density function of θ_i is:

$$P(\theta_i) = \frac{\Gamma(\alpha_i + \beta_i)}{\Gamma(\alpha_i)\Gamma(\beta_i)} \theta_i^{\alpha_i-1} (1 - \theta_i)^{\beta_i-1} \quad (4)$$

where Γ denotes the gamma function. As observations are gathered, and the distribution is updated according to Bayes’s rule. It is particularly convenient to work with Beta distributions because of their conjugacy properties. In particular, each action’s posterior distribution is also beta distribution with parameters that can be updated according to a simple rule:

$$(\alpha_i, \beta_i) \leftarrow \begin{cases} (\alpha_i, \beta_i), & \text{if } a_i \neq i \\ (\alpha_i, \beta_i) + (r_i, 1 - r_i), & \text{otherwise} \end{cases} \quad (5)$$

In other words, we choose region x_i to scan a target address. If we find the target address is active (reward = 1), we will add one to the corresponding α_i (β_i remains unchanged); otherwise (reward = 0), will add one to the corresponding β_i (α_i unchanged). α_i represents the active address probed in region x_i , $\alpha_i + \beta_i$ represents the address budget consumed in region x_i , so the hit rate of active addresses in region x_i is $\frac{\alpha_i}{\alpha_i + \beta_i}$. As in Formula (6), the x_i ’s active address density is proportional to the hit-rate of active address.

$$x_i.\text{density} = \frac{x_i.\text{active addresses}}{x_i.\text{size}} \propto \frac{\alpha_i}{\alpha_i + \beta_i} \quad (6)$$

In active IPv6 address probing, the key issue is to achieve a high active address hit rate within a given budget. Assuming

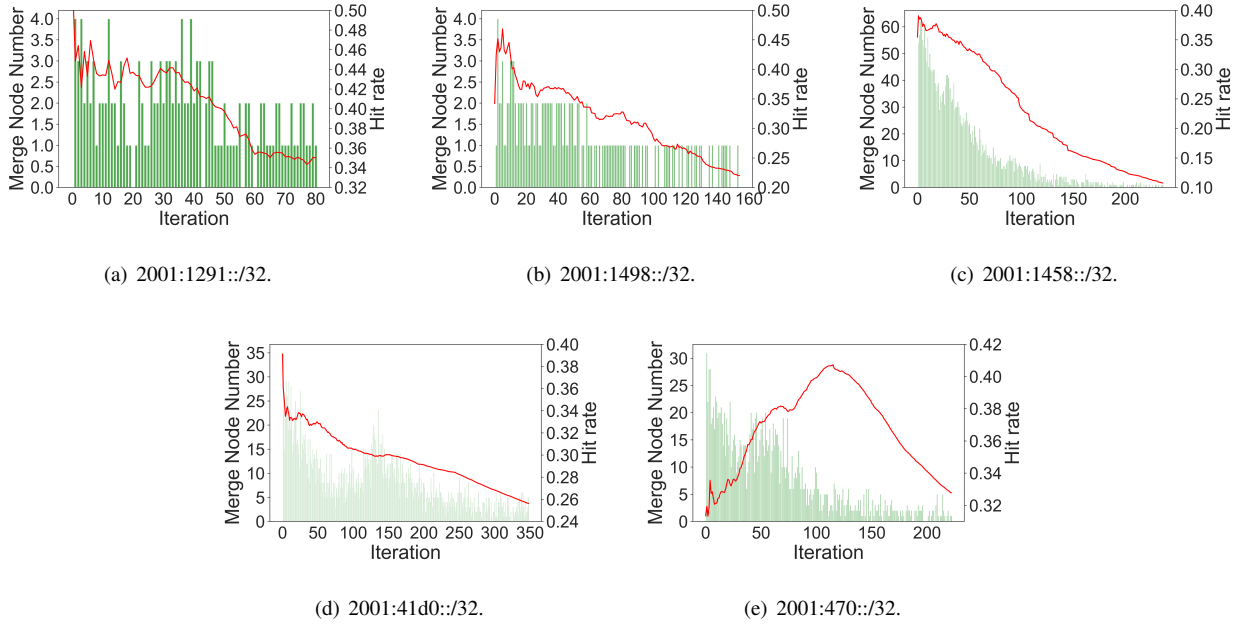


Figure 10: Comparisons of the probing efficiency between the *AddrMiner-S* and other address generation algorithms for different budgets in announced prefixes with sufficient seeds.

that our probing budget is B and the objective function of address probing is f , the active address probing is a combinatorial optimization problem of (X, B, f) , they need to satisfy the following relationship:

$$\sum_{i=1}^n (\alpha_i + \beta_i) \leq B \quad \& \quad \alpha_i + \beta_i \leq x_i.size \quad (7)$$

n represents the number of regions where the target addresses are generated. Our objective function f represents the hit rate of the active address within the target address budget B .

$$f = \frac{\sum_{i=1}^n \alpha_i}{\sum_{i=1}^n \alpha_i + \beta_i} \quad (8)$$

A feasible solution $x \subseteq X$ that satisfies Formula (7), then the most effective solution x^* is only if $f(x^*) \geq f(x), \forall x \subseteq X$.

C Seed Number vs. Probing Efficiency

State-of-the-art address generation algorithms learn the structural and distributional characteristics of seeds to generate target addresses that are more likely to survive. However, these techniques are overly dependent on the quality, quantity, and distribution of seeds. Theoretically, seed address-based target address generation algorithms cannot work in target regions with no seeds, nor can they work efficiently in target regions with few seeds. Next, we further explore the impact of

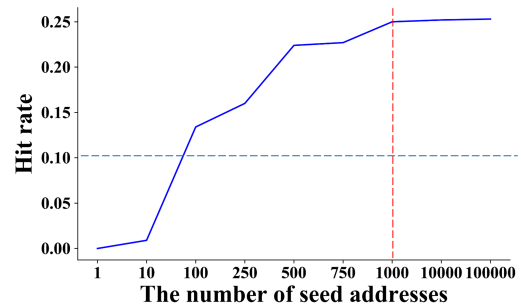


Figure 11: The effect of the number of seeds on the efficiency of active IPv6 address probing.

the number of seeds on the efficiency of active IPv6 address probing.

From the previous measurements, we find that DET [47] has better probing efficiency than 6Tree [33], 6Gen [36], and Entropy/IP [15]. Therefore, we randomly select a different number of active addresses as seeds in any announced prefix with a sufficient number of seeds, and use DET to generate target addresses, with a budget of 10K.

Figure 11 shows the hit rate of active address probing for DET with a different number of seeds. We find that the hit rate of the active address is the lowest when the extreme case of the seed address is 1. At this point, the address probing strategy is the same as 6Tree and fixed-space brute force

Table 6: Overview of our IPv6 Hitlist on September 8, 2021

Name	#IPs	#IPs ¹	#PFXes	#PFXes ²	#Top AS1	#Top AS2	#Top AS3	#Top AS4	#Top AS5
1d-stable	2.1B	1.7B	86.4K	83.8K	20.40%★	16.39%■	13.20%◆	9.45%★	4.65%▶
7d-stable	1.5B	1.1B	85.7K	83.1K	23.41%★	21.48%■	14.44%◆	14.02%★	2.49%■
30d-stable	1.3B	919.4M	80.6K	78.0K	34.96%★	29.75%■	24.05%◆	3.85%★	1.73%■
60d-stable	1.3B	860.2M	80.3K	77.6K	36.74%★	31.83%■	19.62%◆	4.11%★	1.85%■
100d-stable	1.2B	783.7M	80.1K	78.5K	39.58%■	34.93%★	13.58%★	4.52%◆	2.03%■

¹ Removing aliased addresses using aliased prefix detection ★ Amazon, ■ Fastly, ◆ Imperva, ▶ ChinaTelecom, ★ Cloudflare, ■ Akamai.

² Removing aliased prefixes using aliased prefix detection

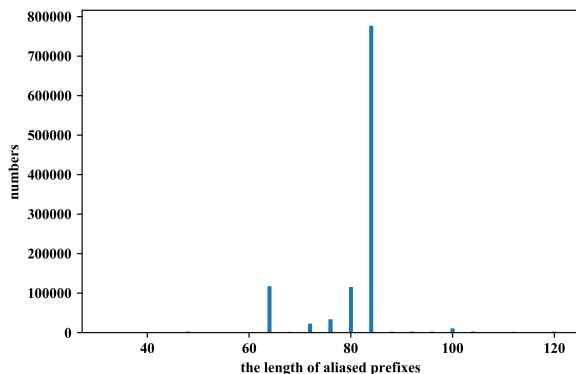


Figure 12: The length distribution of aliased prefixes.

scanning. It builds a hierarchical space tree and prioritizes randomly generates target addresses in low nybble space, so the probing efficiency of this active address is poor (the hit rate is less than 0.01%). As the number of seeds increases, the hit rate of active address probing increases. When the number of seeds in the target region exceeds 100, the active address hit rate exceeds 10%. When the number of seeds exceeds 1000, the hit rate of active addresses remains stable. Therefore, the seed address-based target address generation algorithms are very dependent on seeds and cannot effectively perform global active IPv6 address probing when the number of seeds is insufficient. In this paper, we divide the global active IPv6 address probing task into three sub-tasks according to the number of seeds in the announced prefix spaces, including scenarios with no seeds, scenarios with few seeds, and scenarios with sufficient seeds. We empirically classify the announced prefix spaces. When the number of seeds in the announced prefix space exceeds 1000, we classify the announced prefix space with sufficient seeds. We classify the announced prefix space with few seeds when the number of seeds is not greater than 10. Solutions are designed for each of the above scenarios, effectively solving the global active IPv6 address probing problem.

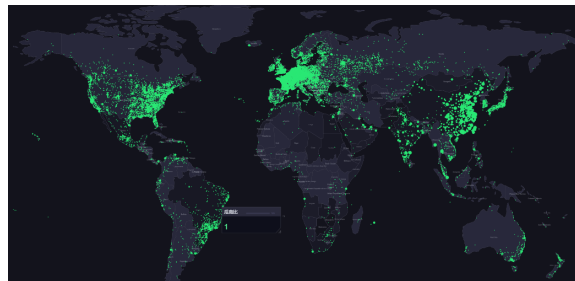


Figure 13: The location distribution of IPv6 hitlist.

D Verification of AddrMiner-S

Figures 10(a) to 10(e) show how the hit rate of active addresses and the number of merged nodes vary with the number of reinforcement learning iterations. *AddrMiner-S* can find more optimal address regions by merging nodes to discover active addresses (exploration) and generate target addresses (exploitation) in high-density regions. When the node space is fully explored, the node merging operation can further expand the address search space. During the exploration process, the hit rate of active addresses does not drop suddenly, ensuring the efficiency of active address probing and discovering new high-density regions. Thus, *AddrMiner-S* strikes a balance between exploration and exploitation. In Figure 10(c), the sampling bias of the seeds is small, so the hit rate is less volatile. In Figure 10(e), the sampling bias is obvious. As the number of iterations increases, high-density regions of the actual network are discovered, which effectively improves the efficiency of address probing.

E Aliased Prefix Analysis

During our evaluation process, we found that aliased prefixes profoundly impact the generation of IPv6 addresses. Because the entire prefix is configured on the same network device, when we do not judge the aliased prefix, all addresses in the aliased prefix space will return an ICMPv6 response packet. These addresses will consume probing resources and cause a lot of false active addresses. Therefore, in our probing system, we consciously detect the aliased prefix and remove aliased

addresses.

Figure 12 shows the length distribution of aliased prefixes. We found that the length of aliases prefix is mostly between /64 and /84. We also surprisedly found that 2,685 aliased prefixes are announced prefixes, such as 2401:5e40:8000::/33, assigned to Japan Network Information Center. We further analyzed and found that the target network opened FTP and Telnet ports and took anti-probing measures. Therefore, we inferred that the device where the prefix 2401:5e40:8000::/33 was located could be a honeypot, a decoy network, or a clustered storage system.

The aliased prefix, especially the entire announced prefix as an aliased prefix, may bring the following security problems: First, heuristic active address probing will guide a large number of probes into the aliased prefix space for probing (if there is no ability to remove the aliased prefix), it will cause excessive load on the network device configured with the aliased prefix, and even multiple normal probers will cause DDoS attacks. In addition, it may also affect the performance of the upper network.

F IPv6 Hitlist Introduction

We further analyze the organizational distribution and address location distribution of the IPv6 hitlist in Table 6 and Figure 13, respectively.



Co-opting Linux Processes for High-Performance Network Simulation

Rob Jansen

U.S. Naval Research Laboratory
rob.g.jansen@nrl.navy.mil

Jim Newsome

Tor Project
jnewsome@torproject.org

Ryan Wails

Georgetown University,
U.S. Naval Research Laboratory
ryan.wails@nrl.navy.mil

Abstract

Network experimentation tools are vitally important to the process of developing, evaluating, and testing distributed systems. The state-of-the-art simulation tools are either prohibitively inefficient at large scales or are limited by nontrivial architectural challenges, inhibiting their widespread adoption. In this paper, we present the design and implementation of *Phantom*,¹ a novel tool for conducting distributed system experiments. In Phantom, a discrete-event network simulator directly executes unmodified applications as Linux processes and innovatively synthesizes efficient process control, system call interposition, and data transfer methods to co-opt the processes into the simulation environment. Our evaluation demonstrates that Phantom is up to $2.2\times$ faster than Shadow, up to $3.4\times$ faster than NS-3, and up to $43\times$ faster than gRaIL in large P2P benchmarks while offering performance comparable to Shadow in large Tor network simulations.

1 Introduction

Network experimentation tools promote the progression of network science: they aim to *realistically* reproduce the effects of distributed networks at *scale* in a *controlled* environment, enabling the scientific evaluation of performance and security across a range of system characteristics. Experimentation tools are particularly useful for *large-scale* distributed systems that are deployed in the real world, such as the globally expansive domain name system [50], peer-to-peer and content distribution networks [14], decentralized data-storage networks [52], and overlay networks [17]. Due to the sizes of these deployments and the internet's great heterogeneity and rapid change [19], it would be extremely difficult to run scientifically controlled, replicable experiments with them in the real world. Tools that enable realistic, scalable, and controlled experimentation of large-scale distributed systems can help accelerate research, development, and education.

Large-scale distributed systems are often characterized by a complex set of algorithms and protocols that run in *application-layer* software. Previous work has found that it is prudent to *directly execute* this software as part of the experimentation process to promote realism [30, 54, 62]. However, there are nontrivial architectural challenges in designing tools that meet the scalability and realism requirements. Emulators such as Mininet [45] do not support large-scale systems because they are vulnerable to time distortion during periods of overload [44]. Simulators such as NS-3 [26] run *application abstractions* in place of real software which can cause unrealistic behavior and lead to invalid results [54].

To meet the large-scale distributed system requirements, the state-of-the-art tools are designed with hybrid architectures wherein a network simulator directly executes application code. However, tools that load and execute applications in *plugin namespaces* (i.e., NS-3-DCE [62] and Shadow [30]) suffer from compatibility and correctness issues and high maintenance costs: applications must be recompiled as plugins, complex code is required to load and run them, and the system calls they make often leak outside of the simulation. On the other hand, tools that run applications as *Linux processes* (i.e., gRaIL [54]) incur considerable inter-process overhead: we have measured at least a $10\times$ performance penalty in running gRaIL due to inefficient process control, system call interposition, and data transfer mechanisms. No existing network simulator simultaneously overcomes the compatibility, correctness, maintenance, and performance challenges found in the state-of-the-art tools.

Introducing Phantom: We present *Phantom*,¹ a novel, multi-process network simulator that: (i) precludes the compatibility, correctness, and maintenance issues that have plagued plugin-based designs; and (ii) overcomes the performance challenges of existing multi-process designs by innovatively synthesizing efficient process control, system call interposition, and data transfer mechanisms. In Phantom, a discrete-event network simulation core directly executes unmodified

Approved for public release: distribution is unlimited.

¹We use *Phantom* as a codename in this paper, but our design is merged into the open-source Shadow simulator and synonymous with Shadow v2 [4].

applications as Linux processes, allowing us to take advantage of native Linux process isolation and management. Phantom co-opts the Linux processes into a simulation environment by (i) preloading a shim library (via `LD_PRELOAD`) that is used to establish efficient mechanisms for process control and function interception; (ii) installing a secure computing (i.e., `seccomp`) filter in the processes to guarantee interposition on system calls that are not preloadable; and (iii) using a novel inter-process memory mapper that allows us to directly read and write process memory without incurring inter-process communication (IPC) overhead. Once the processes are co-opted, Phantom efficiently emulates system calls they make and facilitates communication over a simulated network.

Novel Contributions: This paper makes the following novel contributions to the state of the art in network simulation:

- The innovative design of Phantom, which for the first time shows how to minimize inter-process overhead in a hybrid, multi-process network simulator.
- A high-performance implementation of Phantom.
- An extensive evaluation of Phantom through which we find that it is up to $2.2\times$ faster than Shadow, up to $3.4\times$ faster than NS-3, and up to $43\times$ faster than gRaIL in large P2P benchmarks while offering performance comparable to Shadow in large Tor network simulations.
- A verification of Phantom’s accuracy in small LAN and WAN networks and in large Tor overlay networks.

Impact: This work has high potential for broad impact across multiple communities for the purposes of research, development, and education. First, researchers building software prototypes can use Phantom to quickly evaluate their new distributed system designs in a large-scale network without needing to worry about complicated deployments that are difficult to manage. Second, Phantom can be built into developers’ testing frameworks so that new code can be continuously tested and discovered bugs can be identically reproduced. Third, with facilities to introduce network events (e.g., intermittent delays or failures), Phantom could help teach network and distributed systems courses. The Tor Project has already started using Phantom to develop and test new congestion control protocols before deploying them to the Tor network [57].

Availability: Phantom is merged into the open-source Shadow project as of v2 [4] and our artifacts are publicly available [3].

2 Background and Motivation

We motivate the need for Phantom by identifying the key requirements, existing architectures, and challenges for realistically simulating large-scale distributed systems. (See Appendix A for extended background on related tools.)

2.1 Requirements

Scalability: Recent work finds that it is imperative to run network experiments as close as possible to the deployed scale because reducing the scale can lead to a significant loss

of confidence in the experimental results [40]. Although some statistical confidence can be recovered with repeated trials, it can take many more trials at a smaller scale to achieve the same confidence as larger scale simulations [40].

To increase the scale at which we can run network experiments, a correct and valid execution of the simulation workload should not depend on the computational abilities of, or passage of time on, the host machine. Decoupling the simulation from time and computational constraints allows us to scale without introducing artifacts in the results due to over-provisioning and time-distortion [44].

Realism: Distributed systems are often composed of a diverse set of applications that each contain complex logic. We should directly execute these applications in order to guarantee that our experiments identically replicate their logic and obtain the highest application fidelity possible [30, 54, 62].

Deployed system software is often under active development to fix bugs and develop enhancements. We should execute applications the same way they would be executed in deployment; we should not require recompilation or the maintenance of application patches or abstractions. Running unmodified applications enables us to decouple the application logic and programming language from that of the simulation.

Control: Large-scale distributed systems contain many variables, and changing any one of them can have cascading network effects that can lead to unexpected behaviors or results. We should support deterministic execution to obtain scientific control and to guarantee that the results produced by an experiment can be independently and identically replicated.

2.2 Traditional Architectures

Tools implementing strictly traditional architectures are unsuitable for evaluating large-scale distributed systems with logic primarily contained in *application-layer* software.

Simulation: Network simulators such as NS-3 [26] scale independently of the wall-clock time [67] and offer precise experimental control due to deterministic execution [13]. However, simulators traditionally run application *abstractions* in place of real software which can cause unrealistic behavior and lead to invalid results [54]. As a result, traditional simulators do not fulfill the application realism requirement.

Emulation: Network emulators such as Mininet [45] directly execute applications using real kernel network stacks and therefore offer better application realism. However, emulators lack perfect scientific control due to non-determinism [12]. Moreover, emulators are generally unable to scale independently of computational constraints: if the experiment host machine is overloaded, time distortion will exacerbate reproducibility issues [44]. We confirm this claim with an experiment in which we find that as the host machine becomes more loaded with virtual peers, *its packet forwarding capacity is limited* and a decreasing fraction of the sent packets are correctly forwarded (see §5.4 and Figure 14 for details). As a result, traditional emulators are useful only at small scales.

Table 1: Properties of Network Experimentation Architectures

Architecture	Example Tool	Scalability*	Realism†	Control‡
Emulation	Mininet [45]	○	●	○
Simulation	NS-3 [26]	●	○	●
Hybrid	This Work	●	●	●

* Experiments scale independent of time or computational constraints.

† Unmodified applications can be directly executed without recompilation.

‡ Results can be deterministically replicated with the same RNG seed.

2.3 Hybrid Architectures and Challenges

A hybrid architecture is characterized by the ability to directly execute applications to promote realism while still running them in the context of a cohesive network simulation. As a result, a hybrid architecture enjoys the advantages of both emulation and simulation and offers the best opportunity to fulfill the scalability, realism, and control requirements discussed in §2.1 (see Table 1). However, there are numerous challenges with hybrid architectures that we believe have inhibited tools implementing them from achieving widespread adoption. We describe these challenges by the method for executing applications: *plugin namespaces* and *processes*.

Plugin Namespaces: In this approach, the simulator loads each application into a new plugin namespace (e.g., using `dlmopen`) and directly executes the application in the context of that namespace while using function interposition (via `LD_PRELOAD`) to hook the loaded applications into the simulation environment. A plugin design is implemented in both NS-3-DCE [62] and Shadow [30] and has several limitations:

- *Compatibility:* The domain of supported applications is limited to those that are compiled as position-independent libraries (PIC) or executables (PIE) that export their symbols to the dynamic symbol table (`rdynamic`), are dynamically linked to `libc`, and make all system calls through `libc`. Rebuilding is tedious and impossible if the source code is not available (e.g., closed-source software or malware).
- *Correctness:* Relying solely on preloading is unreliable because only dynamically linked functions (e.g., those in `libc`) can be intercepted using `LD_PRELOAD`; system calls invoked via statically linked code or assembly instructions will leak outside of the simulation and cause errors.
- *Maintainability:* A custom dynamic loader [63] is required to load more than 16 namespaces at once, and a portable threading library [48] is used to support multi-threaded applications (these account for 62k LoC in Shadow; see §4). `libc` functions with nontrivial functionality must be reimplemented in order to intercept the system calls they make.

These challenges have limited Shadow’s use to Tor network simulation [40] while work on simulating Bitcoin has been abandoned [48] and work on NS-3-DCE has mostly stalled.

Processes: In this approach, applications are executed as standard Linux processes and hooked into the simulation through the system call interface using standard kernel facilities. This design overcomes many of the limitations of the plugin ap-

proach: (i) the simulator can execute any existing application without rebuilding it; (ii) kernel subsystems guarantee reliable process isolation and correct system call interception; and (iii) the maintenance of a custom loader, threading libraries, and reimplemented `libc` functions is no longer required. However, the naïve way of connecting multiple processes in a cohesive simulation as demonstrated in gRaIL [54] requires the kernel’s process control (`ptrace`) subsystem and is significantly less performant than the plugin approach: we show in §5.4 that the run time of gRaIL (which extends NS-3) is 13× that of NS-3 alone, and 43× that of Phantom in experiments with fixed P2P messaging workloads. Worse performance in gRaIL’s multi-process design can be attributed to:

- *Process control:* The simulator needs to control the execution state of the processes as they progress through simulated time. The `ptrace` process control mechanism (`PTTRACE_ATTACH` or `PTTRACE_TRACEME`) incurs overhead that is *quadratic* in the total number of attached processes, limiting scalability (see Appendix B.1).
- *System call interposition:* The simulator needs to intercept system calls made in the processes so they can be emulated. The `ptrace` system call mechanism (`PTTRACE_SYSCALL`) requires *at least 4* context switches *for every system call*, contributing substantial overhead relative to a same-process function call (see Appendix B.2).
- *Data transfer:* The simulator needs to access system call arguments referencing process memory (e.g., data buffers). The `ptrace` memory access mechanism (`PTTRACE_PEEK` and `PTTRACE_POKE`) requires an additional system call and mode transition *for each word of memory*, making it inefficient for large structs and buffers (see Appendix B.3).

Ideally, we want a simulator with the higher performance of the uni-process, plugin-based Shadow design (which does not incur inter-process overhead) *and* the improved compatibility, correctness, and maintainability of the multi-process gRaIL design. However, it was previously unknown if this ideal is attainable due to the multi-process challenges; indeed, we show throughout §5 that even a more efficient use of `ptrace` (see Appendix B) is still less performant than a uni-process design.

3 Design

In this section we describe the novel multi-process Phantom design that eliminates the limitations of the state-of-the-art plugin-based architecture and overcomes the performance challenges of the state-of-the-art process-based simulator.

3.1 Overview

The main component in Phantom is a discrete-event simulator which drives the simulation (see Figure 1). After initialization, the simulator directly executes the real applications of an experiment as Linux processes while using inter-process communication channels (IPC) between the application and simulator processes. Phantom co-opts the applica-

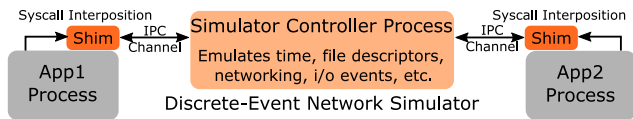


Figure 1: Overview of the Phantom design. Phantom directly executes application processes, intercepting system calls and handling them using a shim and an inter-process communication channel.

tion processes into the simulation by intercepting all system calls they make (e.g., `socket`, `listen`, `connect`, `send`, `recv`, `poll`, etc.) rather than allowing them to be handled by the Linux kernel. Phantom handles intercepted system calls by internally simulating common kernel functionalities that most applications expect to be available, such as networking facilities (e.g., buffers, protocols, and interfaces), event notification facilities (e.g., `select`, `poll`, and `epoll`), and file descriptor facilities (e.g., files, sockets, and pipes). As a result, Phantom emulates a Linux kernel to the applications while connecting them through a virtual, simulated network, and the applications need not be aware that they are running in a simulation.

3.2 Components

3.2.1 Simulation Controller Process

Phantom is a parallel, conservative-time, discrete-event network simulator that emulates a Linux kernel to the applications it executes. Simulations are driven by a single controller process which has two primary functions that occur successively during an *initialization* phase and an *execution* phase.

Initialization Phase: During initialization, the controller reads and processes configuration inputs. The inputs specify a number of virtual hosts that should be simulated, a network graph model that should be used to model network characteristics such as routing, latency, and packet loss between the virtual hosts, and the file paths and arguments needed to directly execute the applications on the virtual hosts. The controller initializes internal simulation state accordingly.

Execution Phase: Simulation work is organized into *events* that each occur at a discrete simulation time. Each event is assigned to a virtual host and stored in a host-specific *event queue*: a min-heap that sorts events by their simulation time.

The controller manages the global simulation clock and synchronizes simulation time by using time barriers to establish discrete *execution rounds*: time intervals during which events may be safely executed in parallel. The time barrier in a round is set such that no event that is executed for any host in that round will enqueue a new event for any other host in the same round. This conservative-time algorithm guarantees that simulation time always advances on each host, even when concurrently executing distinct hosts' events. When the next event time in every host's event queue exceeds the time barrier for the current round, the controller updates the global clock and advances the execution round.

3.2.2 Parallel Worker Threads

Phantom concurrently executes the events in each execution round using *worker threads* (workers) that are managed with high level abstractions we call *logical processors* (LPs). Phantom allows a configurable number of LPs and controls the state of an independently configurable number of workers such that only a number of workers equal to the number of LPs are concurrently active.²

The following algorithm employs a work stealing [10, 65] strategy to schedule the worker threads, ensuring that each LP will always be running a worker thread as long as one with remaining work exists. When an execution round begins, one worker thread starts *running* for each LP while the remaining workers remain *waiting*. While running, a worker dequeues and executes all events that occur within the current round (as set by the controller) for all hosts assigned to it. When a worker completes all outstanding events for the current round, it: (i) starts running another waiting worker that has yet to run in this round (if any exist); and (ii) starts waiting to be run again during the following round. An execution round ends when all workers have entered the waiting state.

3.2.3 Direct Application Execution

During initialization, each virtual host is configured to directly execute some number of applications. Phantom internally creates virtual process and thread data structures to store the state needed to manage the execution of the applications (e.g., file descriptor tables and standard input/output handles). **Managed Processes and Threads:** Phantom directly executes specified application binaries and allows for configuration of the command-line arguments and the start time within the simulation. Each application is launched by a Phantom worker with a `vfork+execvpe` sequence.

The application execution procedure results in the creation of one or more Linux processes and threads that are *managed* by their parent Phantom worker. Each worker (i) uses our preload shim library to co-opt their managed processes into the simulation, and (ii) uses our inter-process communication mechanisms to modulate the running state of the managed processes such that only one of a worker and its managed processes are running at any time (thus maintaining that only one task per LP is concurrently active).

Preload Shim: In order to assist with controlling the managed processes and threads, we create a custom shared library, subsequently referred to as “the shim”, which is loaded into each managed process's address space using the `LD_PRELOAD` environment variable. We use the shim to: (i) execute initialization code in the shim's constructor functions and establish an inter-process communication channel (see §3.2.6); and (ii) intercept functions defined in libraries that are dynamically linked to the applications (e.g., `libc`; see §3.2.4).

²Limiting the number of LPs to be at most the number of available CPU cores avoids performance degradation caused by CPU oversubscription.

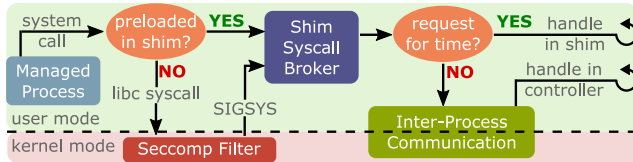


Figure 2: Control flow when intercepting system calls in Phantom.

3.2.4 System Call Interposition

Phantom co-opts processes into the simulation by intercepting functions at the system call interface using two interception strategies: *preloading* and *seccomp* (see Figure 2).

Primary Strategy: Preloading: Recall from §3.2.3 that Phantom preloads a shared library shim into each process it executes using the `LD_PRELOAD` environment variable. Because the shim is preloaded, the dynamic loader loads the shim before all other shared objects linked to the managed process and the shim is the first library searched when attempting to dynamically resolve symbols. This feature allows us to selectively override functions in other shared libraries by supplying identically named functions with alternative implementations inside the shim. Preloading is efficient, as it changes only the address of the instruction that is next executed when a dynamically-linked function is invoked. Therefore, we use preloading as our primary interception strategy.

Notice that preloading works by intercepting *shared library functions*, not *system calls*. While preloading can interpose dynamically linked calls to `libc` system call wrapper functions made from outside of `libc`, it cannot interpose the statically linked calls made from *inside* of `libc` (e.g., internal calls from `printf` to `write`).³ If using preloading alone, we would need to reimplement `printf` and any other `libc` functionality we wanted to support and not just the system call wrappers—an untenable engineering burden. Preloading alone would also fail to intercept system calls made without using `libc` at all, e.g., those made by directly using a `syscall` instruction.

Secondary Strategy: seccomp: Phantom intercepts system calls that are not handled by the preloading strategy using the kernel’s `seccomp` (secure computing) facility. The `seccomp` facility enables a process to set a filter on the system calls that are made by the process and to associate an action with the filter. We install a `seccomp` filter that traps all system calls except for: (i) `sigreturn`; and (ii) system calls originating from Phantom’s own preloaded shim. We install a `SIGSYS` signal handler for system calls trapped by the `seccomp` filter; whenever a system call matching the filter is invoked, the kernel traps it and instead calls our signal handler function.

We use `seccomp` as our *secondary* interception strategy because, although it can intercept all system calls, it is less efficient than preloading; it requires: (i) a mode transition

³APIs that invoke vDSO functions (e.g., `time`) rather than make system calls can either be preloaded or we can dynamically rewrite the vDSO to guarantee that it makes interoperable system calls [55].

from the process to the kernel when the system call is invoked; (ii) execution of the `seccomp` filter; and (iii) a mode transition back to the process to invoke the shim callback function. Because most system calls are preloadable, we infrequently incur the additional overhead from `seccomp` in practice.

3.2.5 Emulating System Calls

Both system call interception strategies from §3.2.4 result in a `syscall` handler function being executed in the shim, i.e., within the managed process. System calls can be emulated either directly in the shim or in the controller (see Figure 2).

In the Shim: Frequently made system calls that can be emulated using little state from the controller can be serviced directly in the shim without incurring additional overhead related to IPC. For example, the shim directly handles the `time`, `gettimeofday`, and `clock_gettime` system calls by arranging for the controller to share and maintain the current simulation time in a shared memory control block that is accessible to the shim as described in §3.2.6.

In the Controller: The remaining system calls are serviced in the simulator controller process. The system call number and arguments are sent to the controller using the IPC control channel as described in §3.2.6. The controller handles the system calls internally using lightweight implementations that effectively form a simulated kernel that completely replaces the functionality normally provided by the Linux kernel. The simulated kernel (re)implements (i.e., simulates) important system functionality, including: the passage of time; input and output operations on file, socket, pipe, timer, and event descriptors; packet transmissions with respect to transport layer protocols such as TCP and UDP; and aspects of computer networking including routing, queuing, and bandwidth limits. (See Appendix D for additional details.) Importantly, this approach enables us to establish a private, simulated network environment that is completely isolated from the real network, but is internally interoperable and entirely controllable.

Determinism: Phantom uses a pseudorandom generator that is seeded with a configurable seed as its single source of randomness throughout the simulation. Care is taken to ensure that all random bytes that are needed during the simulation are initiated from this source, including during the emulation of system calls such as `getrandom` and when emulating reads from files like `/dev/*random`. This approach allows Phantom to produce deterministic simulations, improving scientific control over the experimentation process and enabling experimental results to be replicated.

3.2.6 Managed Process-to-Controller Communication

We use control channels to exchange *fixed-size* messages with each managed process (e.g., system call arguments), and a memory manager to exchange *dynamic* amounts of data (e.g., a buffer passed to a `send` system call; see Figure 3).

Control Channel: Phantom establishes a control channel with the shim of each managed process by allocating an initial block of shared memory and sharing the handle to this

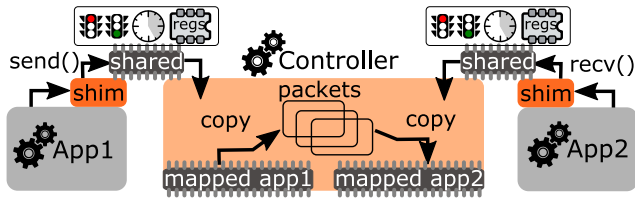


Figure 3: Phantom uses shared memory as a control channel, modulating control using semaphores. The app1 shim intercepts `send()`, writes its arguments into the shared syscall registers, and then uses semaphores to pass control. The controller reads the registers, uses the memory manager to directly copy the send buffer into simulated packets, and schedules an event so the packets arrive at the receiver following network semantics. The controller writes the return register and passes control back so app1 continues running. An analogous process occurs when app2 calls `recv()` (or any other system call).

memory during process startup using an environment variable. This *control block* uses a fixed data structure layout that includes semaphores and messaging state (e.g., system call arguments). The semaphores provide a safe and efficient way for a message sender to signal that a new message is available and for a message receiver to wait for a new message; the controller uses this functionality to modulate the execution state of the process (see §3.2.7). We use shared memory and semaphores because we found this combination to perform better than alternative approaches (see Appendix C).

Memory Manager: We designed an inter-process memory access manager to enable the controller to directly and efficiently read and write the memory of each managed process without extraneous data copies or control messages. The memory manager tracks the memory mappings that are active across various regions of a process’s memory, which are analogous to the mappings found in the `/proc/<pid>/maps` file. Upon initialization, the memory manager creates a sparse memory file for each process, where a virtual address in the process corresponds to the same offset in the file. The memory manager initially *remaps* the process’s stack and heap memory regions into this file. As the process runs, the memory manager brokers all read, write, or other mapping requests that involve managed process memory in order to: (i) also map requests for anonymous private regions (such as those made when serving large allocation requests) into the shared file; (ii) maintain a consistent view of the process’s address space; and (iii) simplify system call handling by translating memory pointers to shared memory pointers as needed. Whenever the memory manager receives an access request for an address that is not mapped into the shared file, it utilizes the kernel’s `process_vm_readv` and `process_vm_writev` facilities to directly transfer data between the controller and the managed process’s address space without copying it into kernel space.

3.2.7 Managed Process/Thread Scheduling

We use the IPC control channel from §3.2.6 to control the execution state of each managed process. When a process

first loads, it immediately waits on the channel semaphore to receive a message from Phantom before starting. When a Phantom worker runs (following the algorithm in §3.2.2), the worker initially sends a start message to the process it manages and waits to receive a message back from the process. The process then runs until it invokes a system call that is interposed as described in §3.2.4, sends a system call request message back through the control channel to the waiting Phantom worker, and waits to receive the system call result message from Phantom.

There are two possible scheduling outcomes when a Phantom worker handles a system call requested by a managed process. For system calls that can be handled immediately (non-blocking calls, or blocking calls for which a result is ready), the Phantom worker returns the result over the control channel and the scheduling cycle continues. For system calls that cannot be handled immediately (blocking system calls whose result is not ready), the Phantom worker must wait for some condition to become true (e.g., a packet to arrive or a timeout to occur). Such conditions are internally registered, and then the worker leaves the managed process in an idle state while it continues executing simulation events (and advancing simulation time). When the condition later becomes true (e.g., a timeout occurred), the worker executes an event that causes it to check the system call state and return the timeout result to the process over the control channel. The process continues executing and the scheduling cycle continues.

The effect of this scheduling process is that each Phantom worker only allows a single thread of execution across all processes it manages; each of the remaining managed processes/threads will always be idle, waiting for a result message from the worker for the previously requested system call. Using this scheduling process, Phantom has precise control over the execution state of all managed processes and guarantees nonconcurrent access of managed processes’ memory through the memory manager from §3.2.6.

3.2.8 Linux CPU Scheduling

Phantom is designed to work with the Linux CPU affinity (i.e., CPU pinning) scheduling feature. CPU affinity is a scheduling attribute associated with running Linux processes. A process’s CPU affinity can be adjusted to restrict the process to run only on a specified subset of CPUs (e.g., a single CPU). CPU pinning can improve performance by reducing the frequency of cache misses, CPU migrations, and context switches. In particular, Linux semaphores shared between two same-core processes incur fewer context switches than when shared between cross-core processes (see Appendix C). Recall that Phantom will run either a worker thread or one of its managed processes, but never both at the same time. This design choice enables us to naturally pin each worker and all of its managed processes to the same core in order to capitalize on the CPU pinning performance benefits.

4 Implementation

We implement Phantom using the plugin-based Shadow as a basis because: (i) we will show in §5.4 that Shadow outperforms other simulators; and (ii) it will be fairer to compare the plugin- and process-based architectures using tools that share the same foundation. See Appendix A.3 for Shadow details.

Transforming Shadow: We forked Shadow v1.14.0 and identified the components that are no longer necessary for Phantom. Of the 94,259 lines of code (LoC) in Shadow v1.14.0,⁴ we removed 47,959 LoC (50.9%) containing a custom version of the GNU portable threads library that was used to simulate application threading [48], 14,498 LoC (15.9%) containing a custom loader that dynamically loads plugins using `dlopen` [63], and 6,559 LoC (7.0%) that implemented the interface between Shadow and the `libc` functions it preloads. We also found that 6,315 LoC (6.7%) implemented tests and 2,123 LoC (2.3%) implemented tools, leaving just 16,805 LoC (17.8%) implementing core simulator functionality that Phantom integrates (see Appendix D for more details).

Implementing Phantom: We implemented Phantom’s design from §3 on top of our stripped down version of Shadow. Our full Phantom implementation supports 164 system calls and contains 56,742 LoC: tests account for about 15,653 LoC (27.6%), tools account for 1,956 LoC (3.4%), and the remaining 39,133 LoC (69.0%) implements core functionality.

5 Evaluation

We evaluate Phantom by running micro- and macrobenchmarks, by verifying its simulation accuracy, and by comparing it to related tools. (See Appendix E for additional details.)

In our benchmarks, we compare three distinct state-of-the-art simulator architectures (see Appendix A): (i) multi-process, `seccomp` (Phantom); (ii) multi-process, `ptrace` (gRaIL); and (iii) uni-process, plugin namespaces (Shadow). For fairness, we compare all three architectures running on top of an identical simulator framework and network stack (i.e., Shadow’s), thus ensuring that we can isolate performance differences and attribute them *exclusively* to the change in architecture and not to, e.g., differently inefficient code running in independent code-bases.⁵

All experiments use CPU pinning and our primary interception strategy (preloading) unless otherwise noted. All simulations were repeated ten times with unique seeds; we present the results as the mean across the ten trials with 99% CIs.

⁴LoC are counted with the `scc` tool: <https://github.com/boyter/scc>

⁵Because gRaIL was originally implemented on top of NS-3, we ported the design to Shadow by implementing `ptrace` as an optional alternative to the `seccomp` secondary interposition strategy. We found and mitigated many sources of `ptrace` overhead (see Appendix B) and our implementation should be considered an optimized, near-best-case version of gRaIL.

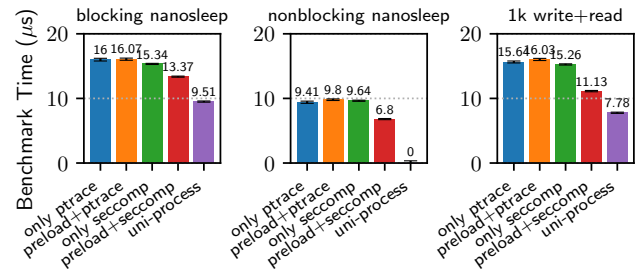


Figure 4: Time to execute blocking, nonblocking, and io-based system calls using several interception methods, compared to Shadow’s uni-process preload-based design.

5.1 Performance: Microbenchmarks

Setup: We anticipate that one of the major sources of overhead in a multi-process design is due to inter-process communication and context switching, which in Phantom occurs whenever a system call is executed. There are three main types of system calls: (i) blocking calls that require the simulator to update state (e.g., advance time) before returning; (ii) nonblocking calls that can return immediately; and (iii) input/output (io) calls that involve reading or writing a dynamically sized buffer. We benchmark these operations using a small program that either invokes the `nanosleep` system call (with a timeout of 1 for blocking or 0 for nonblocking), or invokes a `write` and then a `read` operation on a pipe. The program loops repeatedly for 10k iterations and measures the time required to complete each benchmark after timing 10k iterations of a no-op as a baseline. We report the difference between the mean time to execute each of the three benchmarks and the mean time to execute the no-op baseline.

Results: We ran the benchmarks in our multi-process architecture using several alternative interception methods, and in Shadow’s uni-process preload-based architecture. Figure 4 shows similar trends across all three benchmarks. First, we notice that using preloading and `ptrace` together is slightly slower than using `ptrace` alone; this is because the shim intercepts the system call and then (since it does not have a handler) it invokes the `system` function to pass control to `ptrace`, which adds a few instructions relative to the standard use of `ptrace`. Second, `seccomp` with preloading is significantly faster than `seccomp` alone (and both `ptrace` modes), because preloading allows us to intercept system calls without incurring the overhead of a mode transition and the execution of the `seccomp` filter. Third, the uni-process design is the fastest of all methods tested; while the blocking and io-based system calls incur some overhead due to switching portable threads (using `set jmp` and `long jmp`), a non-blocking system call is effectively a function call.

Figure 5 shows the results from running our io-based benchmark while setting the buffer size to 1k, 4k, 16k, and 64k bytes. Phantom with the relatively simple approach of using `process_vm_readv` and `process_vm_writev` is the

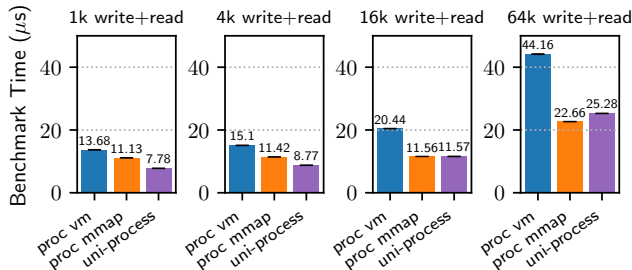


Figure 5: Time to execute io-based system calls using Phantom’s `process_vm_read` and `process_vm_write` fallback facilities compared to its primary inter-process memory mapping design.

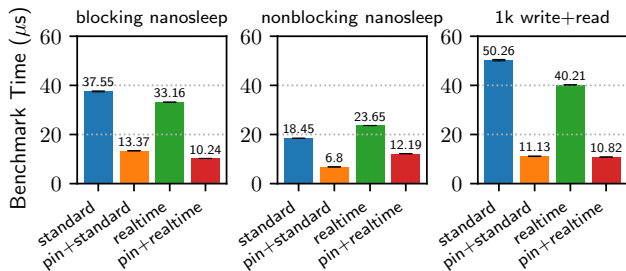


Figure 6: Time to execute our microbenchmarks in Phantom when using the Linux CPU pinning and realtime scheduling features.

slowest. This is partly from the context switch overhead between Phantom and the managed thread for each `read` and `write` call, and the kernel/user mode transition overhead of making the `process_vm` system calls. While these overheads are not dependent on the buffer size, and are amortized for larger buffer sizes, the `process_vm` system calls have significant per-page overhead for validating permissions, pinning each page in memory before doing the copy, and then unpinning them. Hence, the `process_vm` approach gets significantly *worse* than the alternatives as the buffer size increases.

While Phantom’s `mmap`-based approach (§3.2.6) still has the fixed overhead of context switches between Phantom and the managed thread for each `read` and `write`, it uses an interval-map of mapped *regions* (not *pages*) to validate and translate each pointer. Since this cost is fixed, rather than per-page as with the `process_vm` calls, its overhead relative to the uni-process approach is amortized for larger buffers and becomes less than the uni-process overhead for 64 KB buffers.

Figure 6 shows the time to execute our benchmarks using the Linux CPU pinning feature described in §3.2.8 in addition to the `sched_fifo` Linux realtime scheduler. We observe that CPU pinning significantly improves Phantom’s performance while mixed results are obtained when using realtime scheduling. Run time under realtime scheduling decreases by 48–73% when adding CPU pinning, indicating that the primary benefit is from pinning. CPU pinning improves performance particularly well in Phantom due to our design in which a worker modulates its running state and that of each of its managed processes such that no two of these run con-

currently. Therefore, workers and their managed processes will effectively share the same CPU core, improving caching and limiting cross-core migration. We also tested Phantom using `ptrace` and Shadow’s uni-process design and found that pinning provides comparable or better performance than other modes (see Appendix E.1 for more details).

5.2 Performance: Macrobenchmarks

While microbenchmarks enable us to test the effects of system call operations in isolation, macrobenchmarks provide us with a more wholistic understanding of performance while simulating a larger distributed network.

5.2.1 Setup

To run our macrobenchmarks, we write a simple peer-to-peer (P2P) messaging application whose behavior is inspired by the parallel hold (PHOLD) model commonly used to benchmark discrete-event simulators [20].⁶ Our P2P application uses standard UDP sockets for network communication and works as follows. Each peer first creates and sends some number m of messages at startup using `sendto` and then uses `poll` to wait for incoming messages to arrive. Whenever a message is received with `recvfrom`: (i) a number c of AES encryptions and c AES decryptions are performed to produce computational load; and (ii) a new message with a 1k payload is created and sent to produce network load. Whenever a peer sends a message, it makes a weighted choice of the destination peer where peers’ weights are drawn from a configurable probability distribution W ; we use W to create unbalanced workloads across peers.

To benchmark performance we create distributed networks with p peers, each running our P2P application on a distinct virtual host. The network latency between each pair of hosts is set to 50 ms and each host’s bandwidth is unrestricted. All peers start at the same time and run for ten simulated seconds, resulting in 200 communication rounds. Unless otherwise mentioned, our experiments use defaults of $p=1k$ peers, $m=100$ messages, $c=0$ AES (encrypt, decrypt) sequences, and W is the exponential function e^{-3x} for $x \in [0, 1]$ (to produce unbalanced peer workloads).

5.2.2 Results

Interception Strategy and LP Count: We run experiments that vary the interception strategy and LP count to investigate their effects on performance. Our results in Figure 7 show that the uni-process Shadow simulation completes faster than the multi-process Phantom simulations when using 14 or fewer LPs (consistent with our microbenchmark results). However, when the number of LPs exceeds 14, the `seccomp` interception strategy (with or without preloading) performs better than both the `ptrace` strategy and the uni-process design. We observe diminishing returns and performance regressions

⁶We modify the PHOLD model because it is shown to lead to well-balanced workloads that are not representative of real-world networks [11].

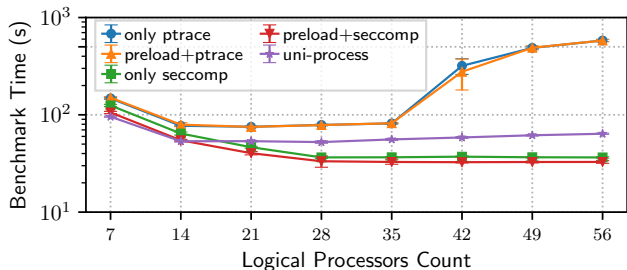


Figure 7: The time to complete our P2P benchmark across a varying set of interception strategies and number of logical processors (i.e. concurrently active worker threads). The y-axis is in log scale.

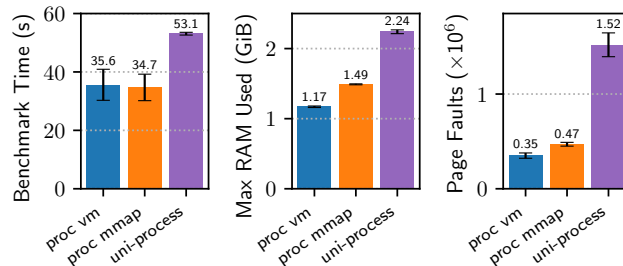


Figure 9: Performance of our P2P benchmark using Phantom’s `process_vm_read` and `process_vm_write` fallback facilities compared to its primary inter-process memory mapping design.

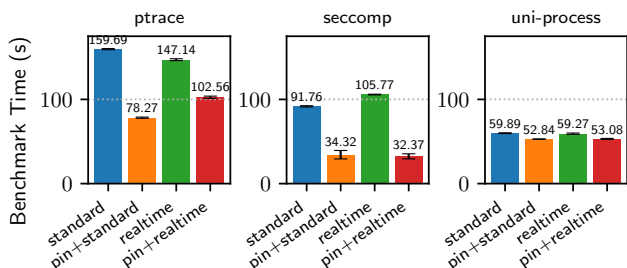


Figure 8: Time to complete our P2P benchmark when using the Linux CPU pinning and realtime scheduling features.

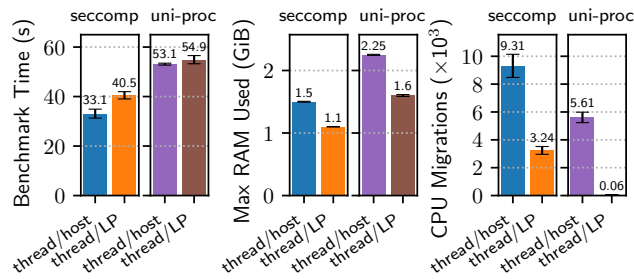


Figure 10: Performance of our P2P benchmark when Phantom (`seccomp`) and Shadow (`uni-proc`) are configured to use one worker thread per virtual host or one worker per logical processor (LP).

with `ptrace` and the `uni-process` design when using more than 14 LPs (the number of cores available on each of the two CPUs), while `seccomp` is able to make effective use of additional LPs. Finally, using a large number of LPs causes more than a $7\times$ slowdown in `ptrace` (from 79s with 14 LPs to 581s with 56 LPs) which we speculate is due to inefficient kernel facilities. We observed similar trends in the initialization time—the time for Phantom to launch all processes or Shadow to load all namespaces: Phantom with `seccomp` is more than $3\times$ as fast (0.84s) as both `ptrace` (3.5s) and the `uni-process` design (2.6s) at initializing processes when using 28 LPs (see Appendix E.2 for more details).

We conclude from our results that using a combination of the preloading and `seccomp` interception strategies leads to the best performance in Phantom: preloading with `seccomp` improves performance over `seccomp` alone because it reduces the overhead from mode transitions and executions of the `seccomp` filter. As in the microbenchmarks, preloading has little effect when used with `ptrace` as expected. Finally, using a number of LPs equal to the number of CPU cores (i.e., half of the available hyper-threads) produces reasonable performance for both Phantom and Shadow. Hence, we use 28 LPs and enable preloading in the remaining experiments.

Linux CPU Scheduling: Figure 8 shows the results of our investigation into the effects of Linux CPU scheduling on the performance of our macrobenchmark. As in our microbenchmarks, we find that CPU pinning has a positive effect on performance: Phantom with `seccomp` completes the benchmark $2.5\times$ faster with CPU pinning than with standard scheduling.

Pinning has a similar but slightly smaller relative effect on `ptrace`, and a positive but minor effect on the `uni-process` design. We observe in our measurements that pinning reduces the number of CPU migrations that occur during the simulation from 5.8M to 8.6k for Phantom with `seccomp`, from 3.6M to 8.1k for Phantom with `ptrace`, and from 180k to 5.7k for Shadow’s `uni-process` design. Realtime scheduling again shows mixed results, but always performs better than standard scheduling when combined with pinning. We conclude that pinning provides a consistently positive effect on performance, and enable it in the remaining experiments.

Inter-Process Memory Manager: Figure 9 shows the performance of Phantom’s inter-process memory mapping design across three metrics. First, we find that Phantom completes the benchmark in comparable time when: (i) using the primary `mmap`-based approach; and (ii) being restricted to the fallback approach of using `process_vm_read` and `process_vm_write`. Recall that our P2P macrobenchmark sends messages with 1k payloads, and in our microbenchmark we found that the performance of the memory mapping approach improves relative to the fallback mechanism as the payload size increases. Second, we observe that the memory mapping design uses slightly more RAM and causes slightly more page faults because it requires additional state to track the memory mappings. Phantom always finishes the benchmark sooner than Shadow’s `uni-process` design ($<70\%$) while using less RAM ($<65\%$) and causing fewer page faults ($<35\%$). (We find that the same general trends hold when running Phantom with `ptrace`; see Appendix E.2 for details.)

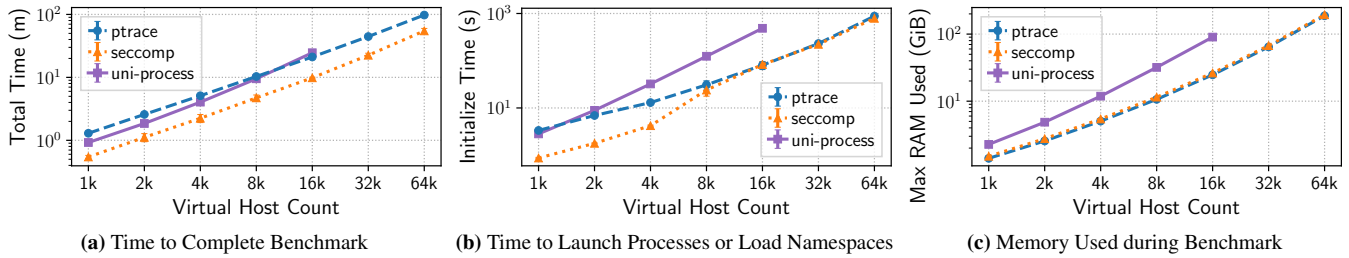


Figure 11: Performance of our P2P benchmark when scaling the number of hosts from 1k to 64k hosts (i.e., processes in Phantom or namespaces in Shadow). Both axes are plotted in log scale on all subplots. Running 32k or more hosts in Shadow exceeded the machine’s RAM (256 GiB).

Worker Thread Scheduling: Figure 10 shows the performance of our work-stealing worker thread scheduling design (described in §3.2.2) in which we run one worker thread per virtual host, compared to a work-stealing algorithm from Shadow that runs one worker thread per logical processor (LP). The benchmark completes more quickly when using one worker thread per host (i.e., 1000 workers in our benchmark) than when using one worker per LP (i.e., 28 workers in this experiment), for both Phantom and Shadow. We observe that by using additional threads, we increase the maximum RAM used and the number of CPU migrations that occur while running the benchmark. (We again find that the same general trends hold when running Phantom with `ptrace`; see Appendix E.2 for details.) We conclude that using more threads should be done in consideration of available RAM.

Peer Workload: We conducted an investigation into the effects of varying peer workloads by varying the number of messages m , the number of AES (encrypt, decrypt) sequences c , and the peer weight distribution W . As expected, increasing m and c resulted in a roughly linear increase in benchmark times in both Phantom and Shadow, while the workload distributions we tested had minor effect on performance. We present more details in Appendix E.2 due to space constraints.

Distributed Network Scale: We investigate the performance of our P2P benchmark while scaling the network size from 1k to 64k hosts. Our results in Figure 11 show that Phantom with `seccomp` outperforms `ptrace` and Shadow’s uni-process design in terms of benchmark time, while initialization time (the time to launch all processes in Phantom or load all namespaces in Shadow) and memory usage both scale more efficiently in Phantom than in Shadow. (Running 32k or more hosts in Shadow exceeded the machine’s RAM (256 GiB).)

Figure 11a shows that the benchmark time exhibits growth that is nearly linear in the number of hosts (`ptrace`: $r = 0.999$, `seccomp`: $r = 0.995$, `uni-process`: $r = 0.994$, where $r = 1$ indicates perfect correlation) with 91ms per host for `ptrace`, 51ms per host for `seccomp`, and 96ms per host for `uni-process`. Accordingly, `uni-process` completes the 16k benchmark in 25m compared to 21m for `ptrace` and 10m for `seccomp` despite running the 1k benchmark in 55s compared to 78s for `ptrace` and 33s for `seccomp`. Phantom with `seccomp` completed the benchmark fastest for all tested network sizes.

Figure 11b shows how the initialization time changes as the host count increases. Here, clear separation between Phantom’s design (which launches multiple processes) and Shadow’s uni-process design (which loads multiple namespaces) can be observed through visual inspection. Despite a slight shift in growth between 4k and 16k hosts for `seccomp`, we find that Phantom is more efficient and scalable than Shadow at initializing virtual hosts’ processes. For example, at 16k hosts Shadow completed initialization in about 8m while Phantom completed it in less than 1.5m.

Figure 11c shows that Phantom uses significantly less RAM to complete the benchmark than Shadow. At 1k hosts Shadow uses 2.3 GiB but Phantom with `seccomp` only uses 1.5 GiB (~65%), while at 16k hosts Shadow uses 90 GiB but Phantom with `seccomp` only uses 26 GiB (~29%). Phantom’s relatively lower memory usage allows us to scale the number of hosts in the P2P benchmark to 4× the size of the largest network in which a benchmark was successful in Shadow.

Conclusions: We draw two primary conclusions from our benchmarks. First, Phantom outperforms the state-of-the-art uni-process Shadow design by effectively mitigating the multi-process performance challenges identified in §2.3. Second, Phantom consistently outperforms a simulator built around our implementation of `ptrace` (which we argue in Appendix B outperforms `gRaLL`’s use of `ptrace`).

5.3 Accuracy: Verification

In this section, we verify that Phantom can accurately simulate basic network characteristics as well as more complex Tor overlay networks [17]. Recall that, as described in §4, Phantom integrates the network stack from Shadow; we do not claim the design or implementation of this network stack as a contribution of this paper. Shadow’s network has already been extensively validated in previous work [30, 32, 37, 40]. Therefore, our primary focus is to verify that Phantom does not *reduce* the accuracy of the simulated network relative to Shadow; we consider our verification successful if Phantom and Shadow produce similar simulated network results.

Basic Network Verification: We evaluate the extent to which Phantom can accurately simulate basic network characteristics that are typical of LAN and WAN networks. Our evaluation considers two nodes that communicate over a single link.

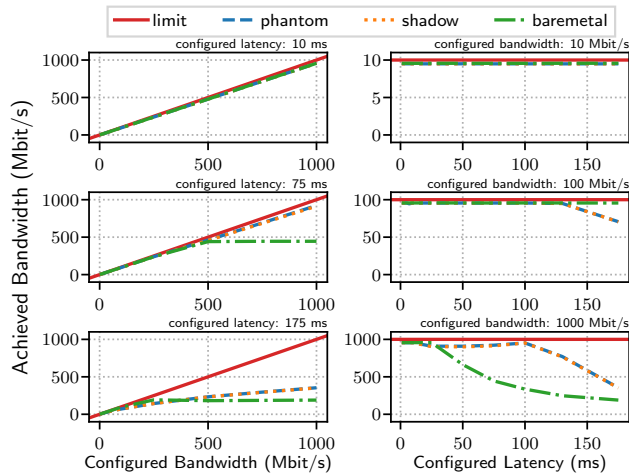


Figure 12: Our basic network verification experiments show that Phantom and Shadow produce identical `iperf` results across a range of configured bandwidths and latencies and that Phantom achieves comparable or higher link utilization than our baremetal setup.

We configure the link with a latency and bandwidth capacity, and then use `iperf` and a UDP ping application to measure the available network bandwidth and latency, respectively, between the nodes. We conducted the experiment across a range of latency and bandwidth settings in Phantom, Shadow, and using two baremetal machines in our lab connected by a 10 Gbit/s physical link (where we used `netem` to emulate the configured latency and bandwidth). Our results in Figure 12 show that: (i) Phantom and Shadow produce indistinguishable results across all tested bandwidths and latencies; and (ii) Phantom generally achieves comparable or higher link utilization than the `netem`-based baremetal setup. (See Appendix E.3.1 for more details, including a description of our latency verification which shows a maximum error of 3%.)

Tor Network Verification: We evaluate the extent to which Phantom can accurately simulate more complex Tor networks using the state-of-the-art Tor modeling tools and methods [40]. We configure a Tor network using a total of 12,232 Linux processes to generate a total of 74 Gbit/s of network traffic, which is equivalent to the expected combined traffic of about 238k users and represents a scale of about 30% of the public Tor network (more explanation is provided in Appendix E.3.2).

We run 10 Tor simulations for 60 simulated minutes each in both Phantom and Shadow and find that: (i) both tools require 27 real hours to run each simulation; and (ii) Shadow uses at most 1116 GiB of RAM while Phantom uses at most 1032 GiB (92.5% relative to Shadow). We measure no significant difference in the simulated network performance across 6 metrics including circuit build time, circuit round trip time, circuit goodput, and Tor network transfer times for 50 KiB, 1 MiB, and 5 MiB files: Figure 13 shows that the performance distributions from Phantom and Shadow are within CI bounds. We conclude that Phantom does not reduce the accuracy relative to Shadow in conducting network experiments.

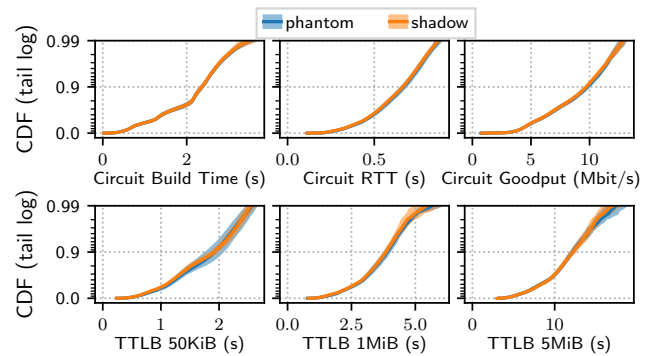


Figure 13: Our Tor network verification experiments show that Phantom and Shadow produce nearly identical Tor performance results (within CI bounds) since they share a network stack.

See Appendix E.3.2 for additional analyses and performance comparisons across a total of 6 network scale factors.

5.4 Comparison to Related Tools

In this section, we compare Phantom to popular tools implementing emulation, simulation, and hybrid architectures.

Mininet: Mininet is a network emulator that creates (i) a network of virtual hosts that run applications as Linux processes; (ii) virtual network interfaces within the Linux kernel; and (iii) virtual switches, controllers, and links that are managed by Mininet [45]. (See Appendix A.1 for more details.)

Mininet’s network emulation architecture offers poor control and scalability because the host kernel is responsible for handling packet events, and the packet routing process is unpredictable and sensitive to load. If the host machine becomes overloaded, Mininet will experience time distortion that will degrade experiment realism and control.

We demonstrate Mininet’s limitations by running a peer-to-peer benchmark (see §5.2.1) while scaling the number of peers in the experiment. Because each peer introduces a constant number of packets into the experiment, the expected number of packets and work to perform in the experiment grows linearly with the number of hosts. However, we find that load and network congestion on the host machine affects the outcome of the experiment. Figure 14 shows the average number of packets received per second by the virtual hosts (averaged over 10 trial runs). As the host machine becomes more loaded with virtual peers, *its packet forwarding capacity is limited*, and fewer packets than expected are forwarded. In contrast, Figure 14 shows that Phantom produces the expected packet throughput in simulated time (Phantom may run faster or slower than real time as necessary to achieve correctness).

NS-3 and gRaIL: NS-3 is a popular network simulator that simulates all aspects of networking *and* all application logic [26], while gRaIL extends NS-3 by enabling simulated nodes to directly execute applications as standard Linux processes managed by the kernel’s `ptrace` facility [54]. (See Appendix A.2 for more details.)

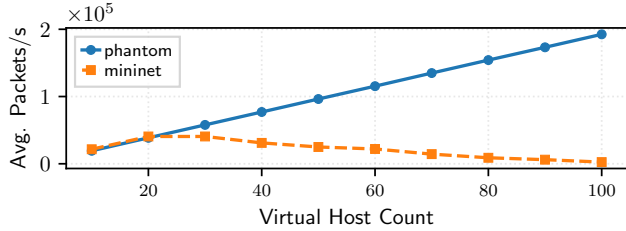


Figure 14: Average packet forwarding rate of a fixed P2P messaging workload in Phantom and Mininet as the number of hosts is varied.

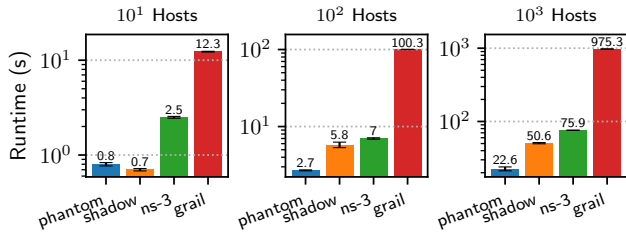


Figure 15: Runtime of a fixed P2P messaging workload in Phantom, Shadow, NS-3, and gRaIL as the number of hosts is varied.

We evaluate the performance cost of running our P2P benchmark from §5.2.1, replicating its logic so that it could also run as an NS-3 application. We configured the benchmark with $m=100$ messages, $c=0$ AES operations, and W =uniform distribution. We vary the number of hosts while configuring 50 ms pairwise latencies and 1 Gbit/s network bandwidths, and run experiments using multiple simulation tools.

Figure 15 shows the real time (mean of 10 trial runs) required to complete 10 seconds of network simulation parallelized across all cores on a blade server (see Appendix E.1) as the number of hosts is varied.⁷ In our 1k host experiment, we find that (i) Phantom is $2.2\times$ and $3.4\times$ faster than Shadow and NS-3, respectively; and (ii) gRaIL’s inefficient multi-process design is about $13\times$ slower than NS-3 alone, and $43\times$ slower than Phantom, demonstrating that Phantom effectively eliminates IPC overhead as a performance bottleneck and overcomes the multi-process challenges from §2.3.

Shadow: Shadow [30] implements a hybrid, uni-process architecture in which applications are directly executed in plugin namespaces and preloading is used (via `LD_PRELOAD`) to intercept `libc` function calls and hook the applications into the simulation. (See Appendix A.3 for more details.)

Although Figure 15 shows that it performs well, Shadow’s plugin architecture is limited in its compatibility, correctness, and maintainability: as shown in Table 2, applications running in Shadow must be compiled as position-independent, must be dynamically linked to `libc`, and must not make system calls via statically linked or assembly code (or else they will not be interceptable). Because we cannot guarantee that

⁷Runtimes are normalized by the amount of work performed (i.e., packets delivered) by each simulator; this resulted in runtime adjustments of $< 5\%$ for all but gRaIL, which delivered only 56% of the expected packets and thus had its runtime adjusted by a factor of 1.8.

Table 2: Application Properties Supported in Hybrid Simulators

Application Property	Shadow	Phantom
Multiple threads (e.g., support for <code>pthread</code> s)	●	●
Multiple processes (e.g., support for <code>fork</code>)	◐	◐
Not position-independent (i.e., PIC or PIE)	○	●
Not dynamically linked to <code>libc</code>	○	●
Symbols not exported to dynamic symbol table	○	●
System calls made in statically linked code	○	●
System calls made in assembly (i.e., avoiding <code>libc</code>)	○	●
100% statically linked (e.g., some go programs)	○	◐

○ Does not work in tool or architecture ● Works in tool & architecture
◐ Not implemented in tool (as of writing) but supported by architecture

`libc` functions will not internally issue multiple unique system calls, Shadow’s design requires reimplementing *both* the kernel system calls *and* the `libc` functions that invoke them.

Phantom overcomes Shadow’s limitations by running applications as standard Linux processes, allowing us to take advantage of the kernel’s high-performance process isolation features. Moreover, Phantom uses `seccomp` to guarantee that system calls can be intercepted no matter how the application initiates them (see Table 2), enabling us to reduce the emulation scope to the system call interface and support a much larger set of applications; Phantom’s supported application set is primarily limited by the system calls and protocols implemented in its simulated kernel, which can be extended over time (our design could also be incorporated into other simulators, such as NS-3). Phantom enjoys these advantages while also meeting or exceeding Shadow’s performance (as we have shown throughout §5).

6 Conclusion

We have designed, implemented, and thoroughly evaluated Phantom, a novel, high-performance network simulator for large-scale distributed systems. Phantom’s multi-process design eliminates the compatibility, correctness, and maintainability limitations that we believe have inhibited the widespread adoption of existing plugin-based simulators. With our innovative synthesis of efficient process control, system call interposition, and data transfer mechanisms, Phantom also overcomes the inter-process performance challenges of the state-of-the-art multi-process simulator. Through our extensive evaluation, we have demonstrated that Phantom achieves better performance and is more scalable than alternative simulators across a variety of important benchmarks.

Acknowledgments: We thank our shepherd and the anonymous reviewers for their valuable feedback. We thank Steven Engler for discussions about design and support during development. This work has been partially supported by the Office of Naval Research (ONR), the Defense Advanced Research Projects Agency (DARPA), and the National Science Foundation (NSF) under award CNS-1925497.

References

- [1] The Tor Metrics Portal. <https://metrics.torproject.org>, April 2021.
- [2] Performance Experiments. <https://gitlab.torproject.org/legacy/trac/-/wikis/org/roadmaps/CoreTor/PerformanceExperiments>, September 2021.
- [3] Artifact for “Co-opting Linux Processes for High-Performance Network Simulation”. <https://netsim-atc2022.github.io>, May 2022.
- [4] Shadow: real applications, simulated networks. <https://shadow.github.io>, May 2022.
- [5] M. AlSabah and I. Goldberg. PCTCP: Per-circuit TCP-over-IPsec Transport for Anonymous Communication Overlay Networks. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.
- [6] M. AlSabah and I. Goldberg. Performance and Security Improvements for Tor: A Survey. *ACM Computing Surveys (CSUR)*, 49(2):32, 2016.
- [7] M. AlSabah, K. Bauer, I. Goldberg, D. Grunwald, D. McCoy, S. Savage, and G. M. Voelker. DefenestraTor: Throwing Out Windows in Tor. In *Privacy Enhancing Technologies Symposium (PETS)*, 2011.
- [8] M. AlSabah, K. Bauer, T. Elahi, and I. Goldberg. The Path Less Travelled: Overcoming Tor’s Bottlenecks with Traffic Splitting. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.
- [9] A. Barton and M. Wright. DeNASA: Destination-Naive AS-Awareness in Anonymous Communications. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2016(4):356–372, 2016.
- [10] R. D. Blumofe and C. E. Leiserson. Scheduling multi-threaded computations by work stealing. *J. ACM*, 46(5): 720–748, Sept. 1999.
- [11] V. Bonnet. Benchmarking parallel discrete event simulations. Master’s thesis, Utrecht University, 2017.
- [12] R. Chertov, S. Fahmy, and N. B. Shroff. Fidelity of network simulation and emulation: A case study of tcp-targeted denial of service attacks. *ACM Transactions on Modeling and Computer Simulation*, 19(1), Jan. 2009.
- [13] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, and G. L. Lee. Determinism and reproducibility in large-scale hpc systems. In *Workshop on Determinism and Correctness in Parallel Programming*, 2013.
- [14] B. Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, 2003.
- [15] B. Conrad and F. Shirazi. Analyzing the Effectiveness of DoS Attacks on Tor. In *Conference on Security of Information and Networks*, 2014.
- [16] S. Dahal, J. Lee, J. Kang, and S. Shin. Analysis on End-to-End Node Selection Probability in Tor Network. In *International Conference on Information Networking (ICOIN)*, 2015.
- [17] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium (USENIX-Sec)*, 2004.
- [18] T.-N. Dinh, F. Rochet, O. Pereira, and D. S. Wal-lach. Scaling Up Anonymous Communication with Efficient Nanopayment Channels. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2020(3):175–203, 2020.
- [19] S. Floyd and V. Paxson. Difficulties in simulating the internet. *IEEE/ACM Transactions on Networking*, 9(4), 2001.
- [20] R. M. Fujimoto. Performance of time warp under synthetic workloads. In *SCS Multiconference on Distributed Simulation*, 1990.
- [21] J. Geddes, R. Jansen, and N. Hopper. How Low Can You Go: Balancing Performance with Anonymity in Tor. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.
- [22] J. Geddes, R. Jansen, and N. Hopper. IMUX: Managing Tor Connections from Two to Infinity, and Beyond. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2014.
- [23] J. Geddes, M. Schliep, and N. Hopper. ABRA CADABRA: Magically Increasing Network Utilization in Tor by Avoiding Bottlenecks. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2016.
- [24] D. Gopal and N. Heninger. Torchestra: Reducing Interactive Traffic Delays over Tor. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2012.
- [25] H. Hanley, Y. Sun, S. Wagh, and P. Mittal. DPSelect: A Differential Privacy Based Guard Relay Selection Algorithm for Tor. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2019(2):166–186, 2019.
- [26] T. R. Henderson, S. Roy, S. Floyd, and G. F. Riley. ns-3 project goals. In *Workshop on NS-2: the IP network simulator*, 2006. See also <https://www.nsnam.org>.
- [27] N. Hopper. Challenges in protecting Tor hidden services from botnet abuse. In *Financial Cryptography and Data Security (FC)*, 2014.

- [28] M. Imani, A. Barton, and M. Wright. Guard Sets in Tor using AS Relationships. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2018(1):145–165, 2018.
- [29] M. Imani, M. Amirabadi, and M. Wright. Modified Relay Selection and Circuit Selection for Faster Tor. *IET Communications*, 13(17):2723–2734, 2019.
- [30] R. Jansen and N. Hopper. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Network and Distributed System Security Symposium (NDSS)*, 2012. See also <https://shadow.github.io>.
- [31] R. Jansen, N. Hopper, and Y. Kim. Recruiting New Tor Relays with BRAIDS. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.
- [32] R. Jansen, K. Bauer, N. Hopper, and R. Dingledine. Methodically Modeling the Tor Network. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2012.
- [33] R. Jansen, P. F. Syverson, and N. Hopper. Throttling Tor Bandwidth Parasites. In *USENIX Security Symposium (USENIX-Sec)*, 2012.
- [34] R. Jansen, A. Johnson, and P. Syverson. LIRA: Lightweight Incentivized Routing for Anonymity. In *Network and Distributed System Security Symposium (NDSS)*, 2013.
- [35] R. Jansen, J. Geddes, C. Wacek, M. Sherr, and P. Syverson. Never Been KIST: Tor’s Congestion Management Blossoms with Kernel-Informed Socket Transport. In *USENIX Security Symposium (USENIX-Sec)*, 2014.
- [36] R. Jansen, F. Tschorsch, A. Johnson, and B. Scheuermann. The Sniper Attack: Anonymously Deanonymizing and Disabling the Tor Network. In *Network and Distributed System Security Symposium (NDSS)*, 2014.
- [37] R. Jansen, M. Traudt, J. Geddes, C. Wacek, M. Sherr, and P. Syverson. KIST: Kernel-Informed Socket Transport for Tor. *ACM Transactions on Privacy and Security (TOPS)*, 22(1):3:1–3:37, December 2018.
- [38] R. Jansen, M. Traudt, and N. Hopper. Privacy-Preserving Dynamic Learning of Tor Network Traffic. In *ACM Conference on Computer and Communications Security (CCS)*, 2018. See also <https://tmodel-ccs2018.github.io>.
- [39] R. Jansen, T. Vaidya, and M. Sherr. Point Break: A Study of Bandwidth Denial-of-Service Attacks against Tor. In *USENIX Security Symposium (USENIX-Sec)*, 2019.
- [40] R. Jansen, J. Tracey, and I. Goldberg. Once is never enough: Foundations for sound statistical inference in Tor network experimentation. In *USENIX Security Symposium (USENIX-Sec)*, 2021. See also <https://neverenough-sec2021.github.io>.
- [41] A. Johnson, R. Jansen, N. Hopper, A. Segal, and P. Syverson. PeerFlow: Secure Load Balancing in Tor. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017(2):74–94, 2017.
- [42] A. Johnson, R. Jansen, A. D. Jaggard, J. Feigenbaum, and P. Syverson. Avoiding The Man on the Wire: Improving Tor’s Security with Trust-Aware Path Selection. In *Network and Distributed System Security Symposium (NDSS)*, 2017.
- [43] K. Kiran, S. S. Chalke, M. Usman, P. D. Shenoy, and K. Venugopal. Anonymity and Performance Analysis of Stream Isolation in Tor Network. In *International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2019.
- [44] J. Lamps, V. Babu, D. M. Nicol, V. Adam, and R. Kumar. Temporal integration of emulation and network simulators on linux multiprocessors. *ACM Transactions on Modeling and Computer Simulation*, 28(1), Jan. 2018.
- [45] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Workshop on Hot Topics in Networks (HotNets)*, 2010. See also <http://mininet.org>.
- [46] D. Lin, M. Sherr, and B. T. Loo. Scalable and Anonymous Group Communication with MTor. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2016(2): 22–39, 2016.
- [47] Z. Liu, Y. Liu, P. Winter, P. Mittal, and Y.-C. Hu. TorPolice: Towards Enforcing Service-Defined Access Policies for Anonymous Communication in the Tor Network. In *International Conference on Network Protocols*, 2017.
- [48] A. Miller and R. Jansen. Shadow-Bitcoin: Scalable Simulation via Direct Execution of Multi-threaded Applications. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2015. See also <https://github.com/shadow/shadow-plugin-bitcoin>.
- [49] A. Mitseva, M. Aleksandrova, T. Engel, and A. Panchenko. Security and Performance Implications of BGP Rerouting-Resistant Guard Selection Algorithms for Tor. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, 2020.
- [50] P. Mockapetris and K. J. Dunlap. Development of the domain name system. In *Communications Architectures and Protocols*, SIGCOMM ’88, page 123–133, 1988.

- [51] W. B. Moore, C. Wacek, and M. Sherr. Exploring the Potential Benefits of Expanded Rate Limiting in Tor: Slow and Steady Wins the Race with Tortoise. In *Annual Computer Security Applications Conference (ACSAC)*, 2011.
- [52] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
- [53] R. Naumann, S. Dietzel, and B. Scheuermann. Towards more realistic network simulations: Leveraging the system-call barrier. In *Ad Hoc Networks*, pages 180–191. Springer, 2017.
- [54] R. Naumann, S. Dietzel, and B. Scheuermann. Push the barrier: Discrete event protocol emulation. *IEEE/ACM Transactions on Networking*, 27(2):635–648, 2019.
- [55] O. S. Navarro Leija, K. Shiptoski, R. G. Scott, B. Wang, N. Renner, R. R. Newton, and J. Devietti. Reproducible containers. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
- [56] J. Newsome (sporksmith). do_wait: make PIDTYPE_PID case O(1) instead of O(n). <https://github.com/torvalds/linux/commit/5449162ac001a926ad8884882b071601df5edb44>, May 2021.
- [57] M. Perry. Shadow Experiments for Congestion Control. <https://gitlab.torproject.org/tpo/core/tor/-/issues/40404>, December 2021.
- [58] F. Rochet and O. Pereira. Waterfilling: Balancing the Tor network with maximum diversity. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017(2):4–22, 2017.
- [59] F. Rochet and O. Pereira. Dropping on the Edge: Flexibility and Traffic Confirmation in Onion Routing Protocols. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2018(2):27–46, 2018.
- [60] F. Rochet, R. Wails, A. Johnson, P. Mittal, and O. Pereira. CLAPS: Client-Location-Aware Path Selection in Tor. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.
- [61] F. Shirazi, C. Diaz, and J. Wright. Towards Measuring Resilience in Anonymous Communication Networks. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2015.
- [62] H. Tazaki, F. Uarbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, and W. Dabbous. Direct code execution: Revisiting library os architecture for reproducible network experiments. In *ACM conference on Emerging networking experiments and technologies*, 2013.
- [63] J. Tracey, R. Jansen, and I. Goldberg. High Performance Tor Experimentation from the Magic of Dynamic ELF's. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2018.
- [64] F. Tschorsch and B. Scheuermann. Mind the Gap: Towards a Backpressure-Based Transport Protocol for the Tor Network. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.
- [65] M. T. Vandevoorde and E. S. Roberts. Workcrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, 1988.
- [66] C. Wacek, H. Tan, K. Bauer, and M. Sherr. An Empirical Evaluation of Relay Selection in Tor. In *Network and Distributed System Security Symposium (NDSS)*, 2013.
- [67] G. Yan et al. Simulation of large scale networks using ssf. In *Winter Simulation Conference*, 2003.
- [68] L. Yang and F. Li. mTor: A Multipath Tor Routing Beyond Bandwidth Throttling. In *2015 IEEE Conference on Communications and Network Security (CNS)*, 2015.
- [69] L. Yang and F. Li. Enhancing Traffic Analysis Resistance for Tor Hidden Services with Multipath Routing. In *International Conference on Security and Privacy in Communication Systems*, 2015.

Appendices

A Background Details for Related Tools

In this appendix we provide extended background on existing tools for network experimentation. We consider popular tools from the architecture categories listed in §2.2 and §2.3: Mininet [45] (emulation), NS-3 [26] (simulation), gRaIL [54] (hybrid, multi-process, `ptrace` controller), and Shadow (hybrid, uni-process, plugin namespaces) [30].

A.1 Mininet

Mininet is a popular network emulator that implements a common design approach for network experimentation tools. Mininet creates a network of virtual hosts that run applications as standard Linux processes, virtual switches that support OpenFlow for custom routing and software-defined networking, and virtual controllers and links. Mininet executes application binaries and routes network packets through virtual network interfaces created within the Linux kernel [45]. The packets are routed by virtual switching and routing appliances managed by Mininet. Network attributes, such as link bandwidth and latency, are emulated using Linux’s traffic control (`tc`) facilities.

Mininet’s design is very flexible: most any application can be run directly without the need of explicitly programmed network simulation routines, making it easy to spin up a new network and quickly start testing software. However, Mininet offers poor control and scalability because the host kernel is responsible for handling packet events, which is unpredictable and sensitive to load as we show in §5.4.

A.2 NS-3 and gRaIL

NS-3 is one of the most widely-used network simulation tools used by network researchers. NS-3 simulations are composed of virtual nodes running application and routing software that are implemented entirely within NS-3’s application logic (written in C++). NS-3 experiments offer a high degree of control and reproducibility, because all aspects of the networking—from generating a packet within an application to physically transmitting the packet’s bits over physical media—are simulated. However, a serious drawback of NS-3’s design is that real applications cannot be run within the simulation. For example, to run `ping` between two NS-3 nodes, a `ping` application simulator must be hand-crafted within NS-3 application code (as opposed to directing the nodes to execute a `ping` binary). Although this requirement may be acceptable to simulate simple applications, generating realistic simulations of complex protocols (e.g., Tor [17]) is difficult due to engineering complexities (e.g., the Tor codebase contains tens of thousands of lines of code).

Two NS-3 modules have been developed that do allow for real application execution within NS-3 simulations: (i) direct-code execution (DCE) mode [62], and (ii) the discrete event

protocol emulation vessel (gRaIL) [54]. In the more recent gRaIL approach, NS-3 nodes are configured to run real application binaries (e.g., `/usr/sbin/ping`) which are forked and executed as genuine Linux processes. The progress of these processes are managed by Linux’s process tracing facility `ptrace`, which allows the NS-3 process to intercept system calls made by the applications and translate them into NS-3 simulation events. Configuring NS-3 with gRaIL improves simulation realism, but has a very high performance cost as we show in §5.4.

A.3 Shadow

Shadow is a hybrid, uni-process network experimentation tool that incorporates aspects of both simulation and emulation [30]. At its core, Shadow is a conservative-time discrete-event network simulator that simulates network protocols (e.g., TCP and UDP), threading (using GNU portable threads [48]), and other kernel operations. Shadow dynamically loads applications into their own namespaces (using `dlopen` and a custom loader [63]) and directly executes application code in the simulator process. Shadow hooks the applications into the simulation using function interposition, but emulates a Linux environment so that application code functions as if it was running in Linux.

Shadow represents the state-of-the-art hybrid network simulator tool for directly executing applications in large-scale distributed system simulations. A primary reason is that Shadow is designed to be high-performance: it runs as a single process (with multiple threads) to avoid inter-process overhead and unnecessary data copies. This has led Shadow to become the standard tool for simulating the Tor anonymity network [40]. However, Shadow has not had widespread use outside of the niche Tor application; Shadow has been shown to simulate Bitcoin networks [48], but that work has since been abandoned due to compatibility, correctness, and maintainability issues as we describe in §5.4.

B Interposing System Calls with `ptrace`

As described in §4, we implemented a system call interposition strategy based on `ptrace` to better understand the performance limits of a simulator designed around `ptrace`. Our `ptrace` implementation provides an alternative to Phantom’s `seccomp` secondary interposition strategy (see §3.2.4).

gRaIL [54] is designed solely around the use of `ptrace` to control processes, system calls, and data transfer. Unfortunately, during our `ptrace` implementation and evaluation, we learned that the way that gRaIL uses `ptrace` (which is a standard and intuitive way to use `ptrace`) results in several scalability and performance problems that significantly reduce the performance of a hybrid network simulator. Since we wanted to understand the performance limits of `ptrace`, we developed enhancements to make `ptrace` more efficient and work around its bottlenecks.

In this appendix, we describe what we learned about making a `ptrace`-based system more performant and scalable. We argue that our improvements make our application of `ptrace` in a simulator significantly more performant and scalable than `gRaIL`'s. Moreover, our implementation inside of Phantom enables us to more fairly evaluate and compare *the best version of gRaIL that it could be* rather than its inefficient prototype.

B.1 Scaling `waitpid` to Many Tracees

In initial evaluations, we were surprised to find that when adding n hosts to a PHOLD [20] simulation, in which the total number of messages passed scales linearly with the number of hosts, the simulation time grew *quadratically* with n instead of linearly. This turned out to be because the `waitpid` syscall, which is used by a `ptrace` tracer to wait for the next `ptrace` stop from a given tracee, performed a linear scan of child tasks, making its performance $O(n)$ in the number of processes in the simulation; i.e. adding a process with some fixed amount of work to the simulation not only added that amount of work, but also made the management of every other process in the simulation slower.

Since this behavior of `waitpid` is potentially surprising, and could hurt performance for other large-scale uses of `ptrace`, we implemented a kernel patch making it $O(1)$ instead of $O(n)$. That patch was accepted, and first included in Linux kernel version v5.13-rc1 [56].

Since we wanted good performance in today's Linux distributions without needing to install a custom kernel, we also implemented a workaround in Phantom's `ptrace` code. Initially we worked around this with a dedicated "fork proxy" thread to initially fork each managed process. This way the (at the time) one-per-CPU "worker thread" weren't parents of the managed processes. However, `waitpid` also performed an $O(n)$ linear scan of *tracees*. This meant that when switching from running one managed thread to another, the worker thread needed to `ptrace-detach` from the blocked thread (sending it a `SIGSTOP` to prevent it from running), and `ptrace-reattach` to the next thread to run. This workaround added substantial overhead, but was an overall performance improvement for simulations involving more than around 1000 managed threads per worker thread.

We were later able to remove the fork-proxy workaround when we moved to the logical-processor-based scheduler described in §3.2.2. Since each worker thread only manages the processes and tasks of a single simulated host, `waitpid` does not become more expensive as hosts are added.

B.2 Reducing Per-syscall `ptrace` Stops

Many `ptrace`-based systems, including `gRaIL` [53, 54] use the `PTRACE_SYSCALL` command to execute the tracee until its next syscall. When the tracee makes a syscall, the tracee is put into a `syscall-enter-stop`. The tracer can then fetch the memory registers of traced program to examine the syscall arguments using a `PTRACE_GETREGS` command (and memory

referenced by those parameters as per Appendix B.3). In the case where the tracer desires to *emulate* the syscall, as is usually the case in Phantom, the tracer can:

1. issue a `PTRACE_SETREGS` command to change the syscall-number being requested to an invalid one;
2. issue another `PTRACE_SYSCALL` command to allow the tracee to execute the syscall, which will result in the issuing of an `ENOSYS` signal that puts the tracee into a `syscall-exit-stop`;
3. overwrite the error result to emulate the original syscall using `PTRACE_SETREGS` etc.; and
4. allow the tracee to continue running again with another `PTRACE_SYSCALL` command.

Using this approach, there are a minimum of 4 context-switches per syscall (assuming the tracee and tracer are executing on the same CPU):

1. tracee to tracer at the `syscall-enter-stop`;
2. tracer to tracee to execute the (no-op) syscall;
3. tracee to tracer at the `syscall-exit-stop`; and
4. tracer to tracee to resume the tracee's execution.

The `ptrace` syscall has an alternative command for when syscalls are to be *emulated* instead of just monitored: the `PTRACE_SYSEMU` command. As with `PTRACE_SYSCALL`, the tracee enters `ptrace-enter-stop` when first encountering a syscall. If the tracer continues again using `PTRACE_SYSEMU`, there is no `syscall-exit-stop`, saving 2 context-switches.

The primary downside of using `PTRACE_SYSEMU` is that if we *really do want* the managed process to execute the original syscall (perhaps with modified arguments), we can no longer just resume the original syscall, because the kernel does not execute the syscall when using `PTRACE_SYSEMU`. In Phantom we instead first get out of the `ptrace-enter-stop`, with a `PTRACE_SINGLESTEP` command, overwrite the instruction pointer to "rewind" it to point to the `syscall` instruction again, and then `PTRACE_SINGLESTEP` again to actually execute it. This adds an extra `ptrace-stop` relative to the case where we would use `PTRACE_SYSCALL`, but this tradeoff is worthwhile when *most* syscalls are being emulated (as is the case in Phantom).

B.3 Efficiently Accessing Tracee Memory

The mechanism for accessing tracee memory via `ptrace` itself is `PTRACE_PEEK` and `PTRACE_POKE`. This is the mechanism used by many `ptrace`-based systems, including `gRaIL` [53, 54] and `DetTrace` [55]. Unfortunately, this mechanism requires a separate syscall and accompanying mode transition to access *each word* of memory, making it inefficient for large structs and buffers.

We could reduce the number of syscalls required for large memory accesses by instead reading and writing the `/proc/[pid]/mem` pseudo-file. After opening the file (which requires already being `ptrace-attached`), accessing a contiguous buffer can be done with a single `pread` or `pwrite` syscall

and multiple buffers can be accessed at the same time with `preadv` and `pwritev`.

However, we instead make use of the `process_vm_readv` and `process_vm_writev` syscalls, which are analagous to `preadv` and `pwritev` but are specialized for accessing the memory of another process. They are a bit simpler to use, since they take the `pid` of the target process instead of needing to open and maintain a file descriptor. They also do not require the caller to be `ptrace`-attached to the target process—a feature that Phantom utilizes in order to use the same code for accessing managed process memory no matter if we are using the `ptrace`- or `seccomp`-based syscall interception strategies.

As discussed in §3.2.6, in Phantom we make most memory accesses even more efficient by remapping some of the tracee’s memory regions into a shared memory file, which is also mapped into Phantom. As we show in Figure 5, this further increases cross-process data transfer performance.

B.4 Enabling Work Stealing

In Shadow, there is roughly one worker thread per-CPU, and in each round of the simulation, each worker thread processes a queue of simulated hosts. When a worker thread’s queue is empty, it *steals* hosts from another worker’s queue.

Unfortunately, when using `ptrace`, only the thread that originally `ptrace`-attached a traced thread is permitted to issue `ptrace` commands; other threads in the process are not. This means that one worker thread cannot steal a simulated host from another worker thread and control its managed processes while the original worker thread is still attached.

We briefly pursued patching the kernel to lift this restriction, but it would involve a fair bit of complexity, and there is understandable hesitancy from kernel developers to add further complexity to the already quite complex `ptrace` subsystem in order to support a somewhat niche use-case.

We initially solved this problem by detaching `ptrace` from each managed thread in a host when done processing that host’s events for the round, and re-attaching each thread the first time we need to control it each round. This process had substantial overhead, which was incurred even if the host and its threads were executed by the same worker the next round.

Phantom’s logical-processor-based scheduler design described in §3.2.2 addresses this problem. In Phantom’s scheduling architecture, worker threads are stolen by logical processors, but hosts never move between worker threads. Therefore a worker thread can stay attached to its managed threads for the entire simulation.

B.5 Avoiding `ptrace` Stops on Some Syscalls

As shown in Figure 4, intercepting a syscall via `LD_PRELOAD` and servicing it via IPC is significantly faster than intercepting and servicing it via `ptrace`. Unfortunately it is difficult to create a hybrid approach that uses `LD_PRELOAD` but falls back to `ptrace`, because `ptrace` stops for *every* syscall. Even if we avoid a `ptrace`-stop by intercepting a

syscall via `LD_PRELOAD`, any syscall we make to communicate with Phantom will still generate a `ptrace`-stop, negating the performance benefit.

We prototyped a solution that worked around this problem by never making a syscall to perform IPC; after sending a message in the shared memory segment, we would do a busy-wait to receive the response instead of making a `futex` syscall. This approach actually worked well for simulations that otherwise had idle cores to spare—the Phantom worker thread could service the syscall on one CPU while the managed thread spun in its loop on another. However, since this effectively halved the maximum concurrency, we ultimately discarded this approach.

A better solution to this problem is to use a `seccomp` filter to have some syscalls generate a `SECCOMP_RET_TRACE` event, and then use `ptrace` to catch those instead of stopping on every syscall [55].

In Phantom we similarly leveraged `seccomp`, but configured `seccomp` to trap to a signal handler (`SECCOMP_TRAP`) in the managed thread. The signal handler uses IPC if needed to communicate with Phantom, doing away with `ptrace` altogether. We have not performed a direct performance comparison between this `SECCOMP_TRAP` approach and the `SECCOMP_RET_TRACE` approach, but expect it to have similar or better performance: while `SECCOMP_RET_TRACE` skips the transfer from control in the kernel to the managed thread before context-switching to the tracing thread, using memory-based IPC from the `SECCOMP_TRAP` handler lets us transfer the register values more cheaply than `PTRACE_GETREGS` and `PTRACE_SETREGS`. More importantly, we can service some syscalls from the `SECCOMP_TRAP` handler without having to context-switch to Phantom at all, as described in §3.2.5.

C Context Switching Performance

In Phantom’s process-oriented architecture, control and messages must be exchanged between Phantom’s worker processes and managed application processes via interprocess communication (IPC). Linux and the POSIX standard offer many facilities for exchanging data across process boundaries and synchronizing processes. In this appendix, we examine the cost associated with a process context switch when facilitated by a given synchronization method. The control flow being measured is as follows, for a communicating parent process and child process pair:

1. The child waits for control from the parent;
2. The parent signals to the child that it should run, and waits for the child;
3. The child gains control, immediately signals back to the parent that it should run, and goes into the wait state; and
4. The parent wakes up and regains control.

In other words, this benchmark measures the latency required to perform a context-switch round-trip from a parent process to a child process and then back to the parent.

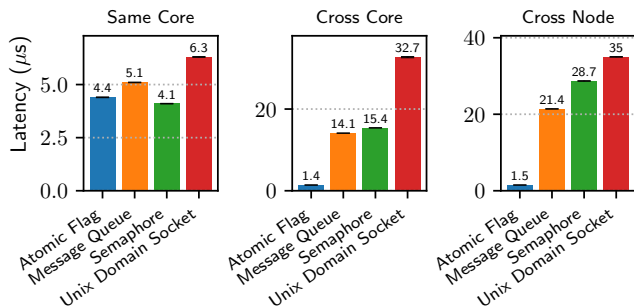


Figure 16: Time required to perform a process context-switch round trip under various synchronization mechanisms and process CPU affinity values.

Four synchronization mechanisms are included in the benchmark:

1. An `_Atomic_Bool` flag placed in shared memory. Waiting is performed by repeatedly checking if the value is `true` (i.e., spinning), and signaling is performed by setting the flag's value to `true`.
2. A POSIX semaphore placed in shared memory. Waiting is performed with `sem_wait` and signaling is performed with `sem_post`.
3. A Unix domain socket with ends shared by the parent and child process. Waiting is performed with `recv` called on the socket's file descriptor and signaling is performed with `send` called on the descriptor.
4. A POSIX message queue shared by the processes. Waiting is performed by calling `mq_recv` and signaling is performed with `mq_send`.

In addition to the mechanism, in this benchmark we also vary the CPU affinity of the parent and child processes. The cost of context switching varies depending upon which processors are executing the processes. The benchmark tests three cases: (i) when the parent and child are pinned to the same CPU core; (ii) when the parent and child are pinned to separate cores on the same CPU; and (iii) when the parent and child are pinned to separate cores on separate NUMA nodes and CPUs.

We ran these benchmarks on a machine with two 14-core Intel Xeon E5-2697 CPUs clocked at 2.60 GHz (the same machine used in our §5 experiments). The machine was running CentOS 7 and Linux kernel version 5.11.6-1. Figure 16 shows the average context-switch round-trip time taken over 100k repeated trials. (99% confidence interval ranges are drawn at the top of each bar, but in every case the interval size is almost zero.) These results show that POSIX message queues and semaphores slightly outperform Unix domain sockets, and that semaphores are the most efficient synchronization method when the processes are pinned to the same core (which is the case in Phantom). Cross-core and cross-node context switching is significantly more expensive than same-core, with the exception of the atomic boolean flag; when using this syn-

chronization mechanism, having two CPUs that can spin in parallel minimizes latency. However, spinning can waste CPU cycles and is not economical when the CPU is under load. Hence, we find that POSIX semaphores are the most performant mechanism to synchronize control between Phantom and its managed, child processes.

D Simulated System and Network Facilities

In this appendix, we describe at a high level the system and network facilities that Phantom simulates. These simulated components are mostly borrowed from Shadow (see §4), but significant attention was required to integrate these with Phantom's new system call interposition, memory manager, and process scheduling interfaces.

Time: As a simulator, Phantom has complete control over simulated time. Attempts by managed processes to obtain the current time are handled by returning the simulated time (relative to a recent epoch) instead. (Because retrieving time is a hot-path function, time-related system calls are handled in the shim as described in §3.2.5.)

Input/Output: Phantom simulates file descriptors and tracks them using a lookup table for each managed process. This ensures a consistent mapping of file descriptor numbers to the internal objects needed to operate on them. Files are simulated internally by using real OS files and translating between the simulated and real file descriptor numbers. Other descriptors can be completely simulated internally, including sockets, pipes, timers, and events. Event notification facilities (e.g., `select`, `poll`, and `epoll`) can also be simulated by tracking the state of each simulated descriptor and triggering callback events when those states change in a way that requires action. Both blocking and non-blocking operations are supported as described in §3.2.7.

Transport: Phantom is a packet-level simulator that implements simulated versions of protocols such as TCP and UDP. Although packet-level semantics are simulated with respect to the associated socket protocols, packet payloads (application data) sent by the managed processes are copied only once into internal buffers. Transferring this data between virtual hosts across the simulated network amounts to transferring the memory address of the original data location, minimizing overhead when transferring simulated packets.

Network: Phantom simulates DNS using a simple name to virtual IP address mapping. Routing is simplified to running shortest path over a configurable network graph to compute end-to-end latency and packet loss. In addition to these network characteristics, packets sent over the simulated network will also be subject to: (i) virtual host bandwidth limits which are simulated using token buckets; and (ii) network queuing semantics which are simulated using an implementation of the CoDel (controlled delay) network scheduling algorithm.

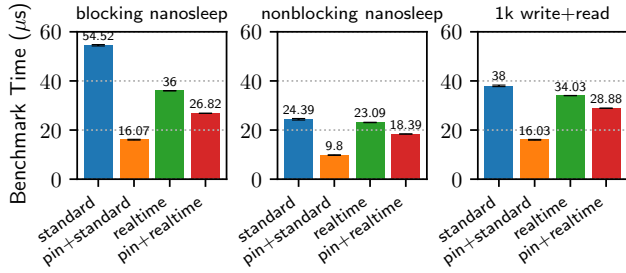


Figure 17: Time to execute our microbenchmarks from §5.1 when Phantom is configured to run with the `ptrace` interception strategy and the Linux CPU pinning and realtime scheduling features.

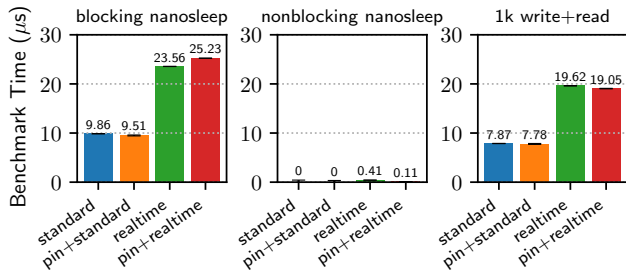


Figure 18: Time to execute our microbenchmarks from §5.1 in the uni-process, preload-based Shadow architecture when using the Linux CPU pinning and realtime scheduling features.

E Extended Evaluation

In this appendix, we include some extended evaluation details and results that we were unable to include in the main body of the paper (in §5) due to space constraints.

E.1 Extended Microbenchmarks

We conducted the evaluation here and in §5.1 using a blade server cluster in which each blade contained identical hardware: 256 GiB of RAM and 2×14 core Intel Xeon E5-2697v3 CPUs (56 total hyper-threads) running at 2.6 GHz. Each blade machine was running CentOS 7 and Linux kernel version 5.11.6-1. We configured our experiments to run in docker containers to ensure that we were running identical software stacks across the blade machines.

Figure 17 shows the effect of Linux CPU scheduling features when running Phantom with a `ptrace` interception strategy. We find that CPU pinning alone performs the best and that adding realtime scheduling along with CPU pinning has an adverse effect. Running with realtime scheduling alone only slightly improves performance over using the standard Linux scheduling mechanisms.

Figure 18 shows the effect of Linux CPU scheduling features on a uni-process design. The choice of Linux CPU scheduling feature has little effect on the nonblocking `nanosleep` benchmark, since it is effectively a function call and already incredibly efficient. Interestingly, realtime scheduling reduces performance for both the blocking and io-

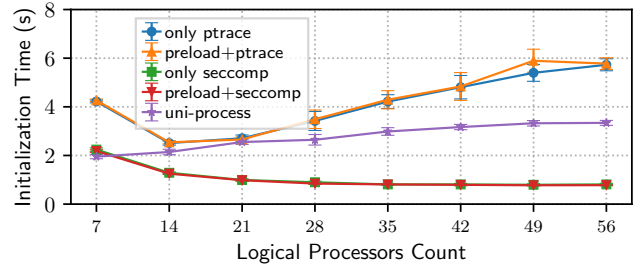


Figure 19: Initialization time is the time for Phantom to launch all managed processes (or for uni-process Shadow to load all namespaces) and run them until the first blocking system call.

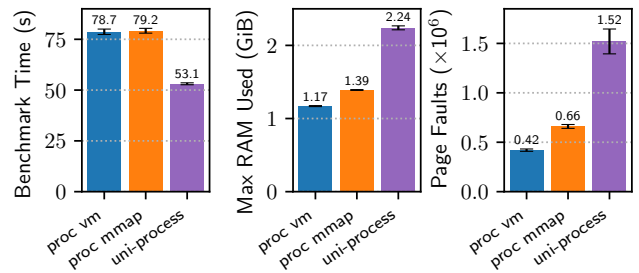


Figure 20: Performance of our P2P benchmark with Phantom using `ptrace` interception with `process_vm_read` and `process_vm_write` fallback facilities compared to its primary inter-process memory mapping design.

based benchmarks, and adding pinning and realtime scheduling together does not mitigate these effects.

E.2 Extended Macrobenchmarks

We conducted the evaluation here and in §5.2 using the same blade server cluster that we used for the microbenchmarks (see Appendix E.1).

Interception Strategies: Figure 19 shows the initialization time for Phantom with various interception strategies compared to the uni-process Shadow design. As expected, preloading has an insignificant effect on launch/load times. When using `seccomp`, Phantom completes the initialization process the faster than the uni-process design. Initialization takes the longest when using `ptrace`, and scales the worst as the number of LPs increases.

Memory Manager: Figure 20 shows that, when configured to use the `ptrace` interception strategy, the inter-process memory mapping design performs comparably to `process_vm_read` and `process_vm_write` fallback facilities despite using slightly more RAM and causing slightly more page faults. Phantom’s memory manager uses less RAM than Shadow’s uni-process design in all cases.

Thread Scheduler: Figure 21 shows the performance of the work-stealing thread schedulers when running Phantom with the `ptrace` interception strategy compared to the performance when using the schedulers in the uni-process Shadow design. Consistent with our results from §5.2, we find that

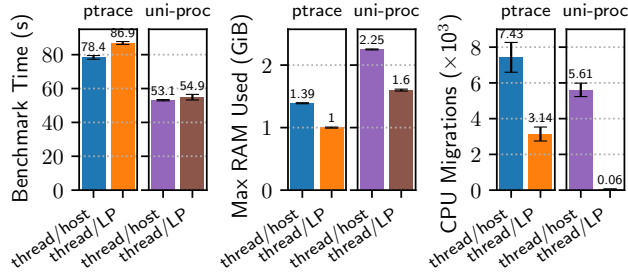


Figure 21: Performance of our P2P benchmark when our worker thread scheduler uses one thread per virtual host or one thread per logical processor (LP).

using one worker thread per virtual host decreases the time to complete the P2P benchmark relative to using one worker thread per LP despite using more RAM and causing more CPU migrations.

Peer Workload: We investigated the performance effect of varying the workloads in our P2P benchmark. In §5.2.1 we described the configurable parameters in our benchmark: each peer sends m messages, when receiving a message a peer performs c AES (encrypt, decrypt) sequences before sending another message, and message destinations are selected according to a peer weighting function W .

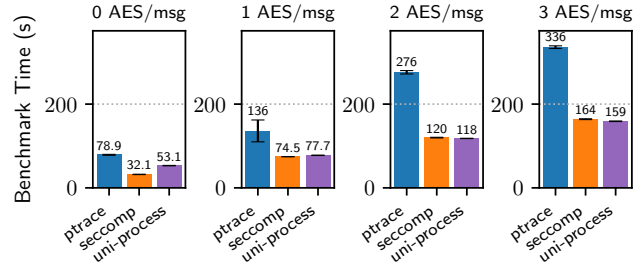
Figure 22a shows the results when varying c in the range $[0, 3]$. In Phantom with `seccomp` interception and in Shadow’s uni-process design, we observe a roughly linear increase in the benchmark time when increasing the number of AES operations that must be performed upon receipt of every message as expected. Interestingly, the linear constant appears to be slightly larger for `seccomp` than for uni-process, and much larger for `ptrace`. We speculate that the observed performance may be due to caching differences.

Figure 22b shows the results when varying m in the set $\{1, 10, 100, 1k\}$. After subtracting the baseline initialization time (represented roughly by the 1 msg/host case), we observe a linear increase in the benchmark time as we increase the message load; this is the expected result since increasing the message load also increases the amount of work the simulator must perform.

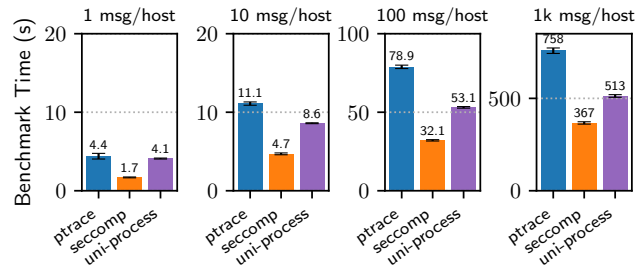
Figure 22c shows the results when varying the workload distribution. In addition to the exponential function defined in §5.2.1 as e^{-3x} for $x \in [0, 1]$, we also consider a “uniform” distribution where each peer has an equal probability of being selected as the destination for any message, and a “ring” distribution where each peer simply selects its closest neighbor as the destination of its outgoing messages. We can see from Figure 22c that these alternative workload distribution functions have minor effect on performance in our simulations.

E.3 Extended Network Verification

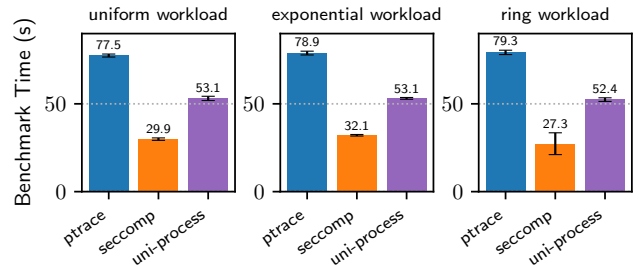
In this appendix, we present extended details and results from our evaluation of Phantom’s ability to accurately represent network characteristics from §5.3.



(a) The effects of varying CPU load



(b) The effects of varying message load



(c) The effects of varying peer load distribution

Figure 22: Performance of our P2P benchmark in Phantom using the `seccomp` and `ptrace` interception strategies and in Shadow’s uni-process design while varying the benchmark CPU and network load parameters.

E.3.1 Extended Basic Network Verification

We verify that Phantom accurately models networked application behavior when those applications are run in different networks with a variety of bandwidth and latency properties. We use two test applications, iPerf and a UDP echo application, to measure Phantom’s simulated bandwidth and latency characteristics. We compare the measurements collected from Phantom to a set of baremetal measurements we collected by running the same applications on two physically-networked servers with emulated bandwidth and latency properties. Additionally, we collected iPerf bandwidth measurements using Shadow so that we could verify that Phantom does not simulate network activity differently than Shadow does. Below we describe the experimental setup and the results from our two experiments.

Setup: To collect simulated bandwidth measurements, we ran a series of iPerf(v2) simulations with Phantom and Shadow. In each simulation, a single client and server communicate

using iPerf’s single-threaded TCP benchmark. The client sends traffic for 10 s, and the server runs until all the data from the client is received. Bandwidth is recorded from the server, which is receiving the traffic, in three-second intervals. Each simulation uses a network configured with a different bandwidth and one-way latency. For these experiments, we consider bandwidth values between 1 Mbit/s and 1 Gbit/s, and one-way latency values between 1 ms and 175 ms. We determined that 175 ms was a realistic upper-bound on latency in wide-area networks by examining RIPE Atlas ping measurements from Jan. 11, 2021: we found that the maximum latency reported by the probes for all built-in measurements was 173.5 ms after removing outliers.

To collect simulated latency measurements, a custom UDP echo application is ran between a client and a server with Phantom. The echo application simply measures the time required for the client to echo a UDP packet sent by the server, which estimates the round-trip time between the server and the client. We consider the same range of latency values in these experiments as in the bandwidth experiments.

To collect emulated baremetal measurements for comparison, we ran the same applications on two physically-networked servers. The machines had 2×Intel Xeon E5-2697 v3 CPUs, 256 GiB of RAM, and ran Debian 11 with Linux Kernel v5.10.0-8. They were both connected with NetXtreme II BCM57810 10 Gigabit Ethernet NICs through a 10 Gbit/s switch. We used Linux’s `netem` facilities to configure each machine’s NIC with a specified bandwidth rate limit and packet latency. Additionally, we set the machine’s TCP stack to use the Reno congestion control algorithm, which is also implemented in Shadow and Phantom.

iPerf Bandwidth Measurements: Figure 12 in §5.3 compares iPerf-reported bandwidth from the baremetal measurements, Phantom, and Shadow. In the left three plots, latency is held constant (at either 10 ms, 75 ms, or 175 ms) and iPerf-reported bandwidth is plotted versus the experimentally-configured bandwidth limit. In the right three plots, bandwidth is held constant (at either 10 Mbit/s, 100 Mbit/s, or 1 Gbit/s) and iPerf-reported bandwidth is plotted versus the experimentally-configured packet one-way latency. These plots show the bandwidth reported by iPerf during the 3-second interval closest to the half-way point of the transfer, which estimates the sustained maximum bandwidth achieved between the client and the server. We find that Phantom does not change modeling accuracy relative to Shadow, and that Phantom-reported performance matches baremetal-reported performance in most network conditions. With more extreme bandwidth and latency values, Phantom is able to achieve higher link-utilization than baremetal. These differences may be accounted for by different parameterizations of Phantom’s TCP stack and Linux’s (e.g., different initial window sizes). Extending and tuning Phantom’s TCP stack to more closely approximate the behavior of Linux’s networking facilities is a promising direction for future work.

Table 3: The Number of Virtual Hosts, Processes, and the Amount of Traffic in each Simulated Tor Network of the Given Scale

Network Scale	5%	10%	15%	20%	25%	30%
Clients	436	871	1307	1742	2178	2614
Relays	349	694	1039	1385	1732	2076
Servers	40	79	119	158	198	238
Total Virtual Hosts	825	1644	2465	3285	4108	4928
Tor	785	1565	2346	3127	3910	4690
OnionTrace	785	1565	2346	3127	3910	4690
TGen	476	950	1426	1900	2376	2852
Total Processes	2046	4080	6118	8154	10196	12232
Simulated Gbit/s*	12	24	37	49	62	74
Equivalent Tor Users	39.6k	79.2k	119k	158k	198k	238k

* Mean across 20 total simulations for each network scale.

Latency Measurements: Both the Phantom and baremetal measured RTT latency values matched nearly identically with the value specified in the experiment for all configured latencies. The largest percent-error between the simulated and emulated results (taking the emulated measurements to be ground-truth) was 3%, which occurred at the lowest configured one-way latency (1 ms): the Phantom-measured RTT was 2 ms, whereas the baremetal measured RTT was 2.07 ms. The difference can be accounted for by software processing time, which Phantom does not simulate.

E.3.2 Extended Tor Network Verification

We consider large-scale Tor network simulation as a practical use case for Phantom. By supporting Tor, we believe Phantom will have broader impact particularly among researchers in the privacy-enhancing technologies community since they commonly use simulation [5, 7–9, 15, 16, 18, 21–25, 27–29, 31, 33–37, 39, 41–43, 46, 47, 49, 51, 58–61, 64, 66, 68, 69] to explore Tor performance and security research problems [6]. Moreover, the Tor Project has recently adopted the use of simulation in their own network planning and performance analyses [2], and Phantom has already been used to guide these efforts [57].

The Tor anonymity network [17] contains over 6k relay nodes [1] and about 800k users that are simultaneously active, i.e., running a Tor client and generating network traffic [38]. Constructing a simulated Tor network that is representative of the real Tor network involves a significant modeling and configuration effort [40]. Important factors that must be considered include the number of virtual hosts running Tor clients and relays, traffic generator clients and servers, the network latency between these hosts, the bandwidth available to these hosts, and the configured behavior of the clients, relays, and traffic generators. Fortunately, recent foundational work on Tor network experimentation has contributed methods and tools to guide our experimentation process [40]. We use these tools directly to create Tor network configs and run experiments in both Phantom and Shadow (Tor network modeling is outside the scope of this paper).

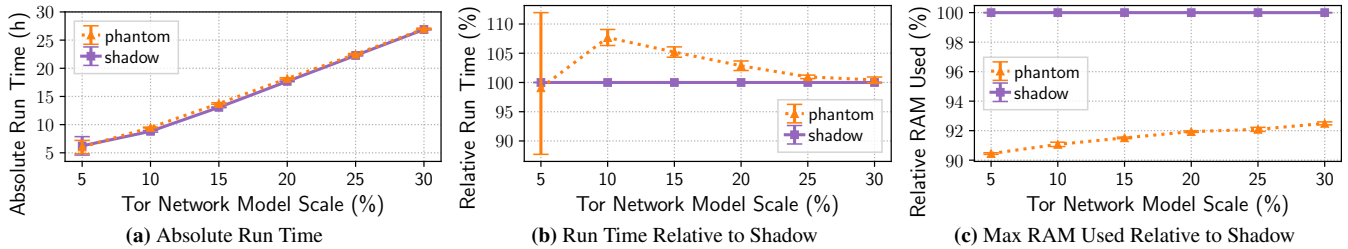


Figure 23: The time and memory required to complete each Tor network simulation in Phantom (using `seccomp` interception) and in Shadow’s uni-process design as the network model scale increases. (b) and (c) show performance relative to Shadow’s baseline.

Setup: We generate 10 unique Tor network configs for each of 6 network scale factors using Tor network state from 2021-01; Table 3 shows the total number of virtual hosts and Linux processes used at each scale. Each client host runs three processes: (i) a TGen traffic generator process that generates traffic according to Markov models created by measuring the real Tor network [38]; (ii) a Tor process in client mode that forwards the TGen traffic into our private Tor network; and (iii) an OnionTrace process that connects to Tor to gather statistics and log information. Each relay host runs a Tor process in relay mode that forwards traffic in our private Tor network, and an OnionTrace process that gathers statistics and logs. Each server host simply runs a TGen server process that coordinates with TGen clients to generate traffic. Table 3 also shows the total volume of traffic being simulated at each scale, and the equivalent expected number of Tor users that it would take to generate that traffic in the public Tor network.

We run each of the resulting 60 Tor networks using Tor v0.4.5.9 in Phantom (using `seccomp`) and in Shadow (the state-of-the-art Tor network simulator). We conducted the evaluation using a blade server cluster in which each blade contained identical hardware: 1.25 TiB of RAM and 4×8 core Intel Xeon E5-4627v2 CPUs (without hyper-threading support) running at 3.30 GHz. For all experiments, we enable CPU pinning, disable realtime scheduling, and use 32 LPs (one LP per core) following our results from §5. We present the results as the mean across the ten networks at each scale with 95% confidence intervals (CIs).

Verification Results: We analyze the performance characteristics in each simulation, e.g., the simulated time to transfer data through the simulated Tor network. Recall from §5.3 that our primary goal is to validate that Phantom does not *reduce* the accuracy of the simulated network stack that it integrates and that we consider our validation successful if Phantom and Shadow produce similar simulated network results.

Figure 13 in §5.3 shows the simulated Tor performance results from using both Phantom and Shadow to each simulate the ten 30% scale Tor networks. Shown are several Tor performance metrics, including: circuit build times; circuit round trip times (time from data request to first byte of response); circuit goodput (transfer rate for range [0.5 MiB, 1 MiB] over 1 MiB and 5 MiB transfers); and client download times for

transfers of 50 KiB, 1 MiB, and 5 MiB. The shaded areas represent 95% confidence intervals that were computed following recently published methods [40]. We do not find a significant difference in performance measured in the simulated Tor network across several metrics when comparing Phantom to Shadow. Again, this is the desired and expected result since both Shadow and Phantom share a network stack implementation. We conclude that Phantom maintains the same level of simulator accuracy that Shadow provided; i.e., Phantom’s multi-process design does not degrade the level of accuracy that can be provided by a simulated network.

Performance Results: Figure 23 shows the simulators’ performance when running the Tor network simulations.

Figure 23a shows that the absolute time to complete a 60 simulated minute experiment is roughly linear in the network scale, where Phantom completed the experiment in the 10%, 20%, and 30% networks in 9, 18, and 27 hours, respectively. Phantom’s run time is remarkably similar Shadow’s, even though Phantom incurs IPC overhead while Shadow does not; this demonstrates the efficiency of Phantom’s design even when simulating somewhat complex distributed systems.

We plot simulation run time relative to Shadow’s baseline in Figure 23b. Although there is large uncertainty in small network scales (consistent with prior work [40]), we observe that Phantom is competitive to Shadow’s uni-process performance at smaller scales and comparable at larger scales: at 30% scale, Shadow’s performance falls within Phantom’s 95% CI.

Figure 23c shows the max RAM used by Phantom to simulate each network scale relative to Shadow’s baseline. We observe that Phantom is more memory-efficient than Shadow: while Shadow used 178, 357, 540, 727, 919, and 1116 GiB at network scales of 5%, 10%, 15%, 20%, 25%, and 30%, respectively, Phantom used at least 90.4% (at 5% scale) and at most 92.5% (at 30% scale) of RAM relative to Shadow.

We conclude that Phantom effectively overcomes the multi-process performance challenges identified in §2.3: Phantom’s multi-process design offers comparable performance to Shadow’s uni-process design while being more memory efficient. In addition to its performance, Phantom offers significant improvements over Shadow because its multi-process design precludes Shadow’s compatibility, correctness, and maintainability limitations that we identified in §2.3.

KSG: Augmenting Kernel Fuzzing with System Call Specification Generation

Hao Sun
Tsinghua University

Yuheng Shen
Tsinghua University

Jianzhong Liu
Tsinghua University

Yiru Xu
Tsinghua University

Yu Jiang^{*}
Tsinghua University

Abstract

Kernel fuzzing is a dynamic testing technique that has successfully found numerous kernel vulnerabilities. However, existing kernel fuzzers, such as Syzkaller, depend on system call specifications to generate test cases. Writing such specifications requires an immense amount of domain knowledge while being extremely laborious. Meanwhile, automated generation of the specification is still an open problem due to the complexity of the kernel, including entry function extraction and input type identification. As a result, the current amount of system call information is insufficient to test the entire kernel code base thoroughly. Syzkaller covers an average of 38% of Linux kernel code with current Syzlang specifications for a prolonged time of fuzzing.

In this paper, we propose KSG to generate system call specifications for kernel fuzzers automatically. First, it utilizes probe-based tracing to extract entry functions accurately. Then, it uses path-sensitive analysis to collect precise input types and range constraints in each execution path of entry functions. Based on the aforementioned information, KSG generates specifications in the domain language Syzlang, which is used by most kernel fuzzers. We evaluated KSG on several versions of the Linux kernel. It automatically generated 2433 unique specifications. Leveraging the newly generated specifications, Syzkaller and Moonshine achieved coverage improvements of 22% and 23% respectively. Furthermore, our approach assisted fuzzers to discover 26 previously unknown bugs, where 13 and 6 bugs were fixed and assigned with CVEs, respectively.

1 Introduction

The operating system kernel is one of the most complex components and forms the foundation of the software system. It is responsible for core functionalities, such as communication and IO of userspace applications. The security of the kernel is crucial as kernel bugs can lead to huge impacts easily, e.g.,

causing userspace applications to be unresponsive [24] and allowing an attacker to completely compromise a target system [22]. Fuzz testing [19] is a popular technique for automatically discovering security vulnerabilities and has already been applied to the kernel domain. For instance, Syzkaller [33], one of the most widely used kernel fuzzers, has been integrated into Linux testing pipeline. It has reported thousands of kernel bugs [31] up until now, demonstrating the effectiveness of applying fuzzing to kernel testing.

The driving force of the kernel fuzzer's bug discovery capability is system call specifications, which provide a rich set of system call information. As shown in Figure 2, a system call can accept parameters with different types based on underlying submodules' requirements. The prototype of system calls are written in C, a weakly typed language, that does not provide much information of system calls' arguments, e.g., many parameters are defined as *void**. Therefore, it is difficult for fuzzers to generate inputs that satisfy the submodules' structural constraints without additional information, resulting in low fuzzing efficiency. To address this issue, existing kernel fuzzers, such as Syzkaller, use a domain language called Syzlang [34] to encode system call specifications. Syzlang is a strongly typed language and can specialize system calls to specific submodules with precise input types, as shown in Figure 1. With this domain knowledge, fuzzing efficiency can be improved significantly. Meanwhile, the amount of specifications has a significant impact on the performance of fuzzers. Fuzzers can generate inputs to test kernel submodules that are well-encoded in the specifications, but they have difficulty reaching kernel code that is not encoded. Therefore, many researchers have to manually write numerous specifications to test the kernel thoroughly.

However, encoding system call specifications requires an immense amount of domain knowledge, resulting in significant time costs and insufficient number of specifications. Many system calls in the Linux kernel are an abstraction over corresponding functionalities of kernel submodules, which are responsible for dispatching the user input to submodules' operations. The actual types of the system call's param-

^{*}Yu Jiang is the corresponding author of this paper.

ters depend on the specific invoked submodule. Therefore, most specifications are encoded for specific submodules, e.g., *dev_loop.txt* in Syzkaller is specifications for *loop* device. In order to encode specifications manually, the following different aspects of domain knowledge are required: (1) the precise input types of system call used by the submodule; (2) the range constraints on specific input parameters; (3) the domain language used by fuzzers for specifications. However, the large amount of kernel submodules and the complexity of the input types make it difficult to encode specifications for a wide range of kernel functionalities. Consequently, most specifications are written by kernel experts, but the extensive manual effort required has caused specification shortage. Based on the coverage data reported by Syzbot's dashboard [32], Syzkaller covers an average of 38% of Linux kernel code with the current Syzlang specifications for a prolonged time of fuzzing. Several recent works perform automatic specification generation [4, 6, 11], but they are either designed for particular system calls or close-sourced scenarios. In order to further reduce manual efforts, we need to generate specifications for more system calls and their corresponding submodules.

Source code analysis can be used to generate specifications effectively, but several challenges need to be addressed due to the complexity of the Linux kernel. First, extracting entries that should be analyzed is difficult because entries can be registered dynamically. In order to generate specifications for specific submodules, we need to analyze submodules' operations invoked by system calls, and we refer to these operations as *entries*. However, entries can be registered dynamically in many scenarios, for instance, during kernel initialization and module loading. Consequently, it is challenging to extract entries using current static analysis methods. Second, the input types of entries can vary in different execution paths, resulting in difficulties identifying them. The functionalities of kernel submodules are complex, while the number of entries for accessing them is limited. In consequence, the submodules' entries can accept different input types across execution paths. We need to collect input types and corresponding range constraints in each execution path of every entry, which poses significant complexity to the analysis. Finally, to generate specifications in domain languages used by kernel fuzzers, we need to perform syntax mapping and semantic encoding based on the collected information.

To address the aforementioned challenges, we propose KSG (Kernel Specification Generator) to automatically generate system call specifications for kernel fuzzers. KSG mainly contains three steps. First, in order to extract submodules' entries without being bound to their implementation details, KSG utilizes a probe-based tracing with Linux *eBPF* [2] and *kprobe* [13]. Based on the extracted entries, KSG uses path-sensitive analysis to collect precise input types and range constraints in each execution path of entries, which is based on Clang Static Analyzer (CSA) [20]. Finally, based on the gathered information, KSG generates system call specifica-

tions in domain language Syzlang, which is used by most kernel fuzzers, to improve the fuzzing efficiency. We evaluated KSG on multiple versions of Linux kernel. It generates 8 specialized calls per minute, with a total of 2433 specialized calls generated in 5 hours, 1460 of which are new to existing specifications. Leveraging the generated specifications, Syzkaller and Moonshine's [23] coverage are improved by 22% and 23%, respectively. Furthermore, KSG assisted fuzzers to discover 26 previously unknown bugs, with 13 and 6 were fixed and assigned with CVEs, respectively.

Overall, we make the following technical contributions:

- We propose an analysis approach for system call specification generation. It incorporates multiple techniques to precisely collect submodules' entries and their types and range constraints on each execution path.
- We designed and implemented KSG, which extracts submodules' entries with *eBPF* and *kprobe* and performs path-sensitive analysis based on Clang Static Analyzer. Leveraging the collected information, KSG generates specifications in Syzlang for kernel fuzzers.
- The evaluation result shows that KSG generated 2433 specifications in total, which can improve the coverage of Syzkaller and Moonshine by 22% and 23% respectively, and assisted fuzzers to find 26 new bugs.

2 Background and Related Works

2.1 Kernel Fuzzing

Fuzz testing is an automated vulnerability discovery approach. Its idea is to continuously generate input to trigger program crashes with the assistance of various sanitizers [27, 28]. Within each fuzz loop, the fuzzer selects the seed from the given corpus with certain guided strategies [5, 25]. Then, it mutates the seed by combining multiple mutation operators, and feeds the generated input to the target program for execution [7, 16]. Meanwhile, the fuzzer collects interesting program behavior, e.g., coverage [3], to determine whether an input is valuable for further mutations, thus continuously optimizing the fuzzing campaign. Take AFL [15] for instance. It is a coverage-guided userspace program fuzzer that has found hundreds of vulnerabilities in widely-used libraries. Many works optimize each part of the fuzz loop [1, 17, 18, 36, 40]. For instance, RIFF [37] moves computations done originally at runtime to instrumentation time, thus reducing the instrumentation code while utilizing vector instructions to improve throughput. Consequently, its coverage measurement mechanism can reduce fuzzing overhead significantly.

The overall process used in kernel fuzzing is similar to that used in userspace, but each specific part can be different due to the complexity of the kernel. Specifically, the idea of kernel fuzzing is to generate high-quality input to trigger

kernel crash assisted with kinds of kernel sanitizers [8, 9], which is the same as a standard fuzz loop. However, unlike userspace fuzzing, the input structure of system calls can be complicated, and the cost of each test case execution is expensive. Therefore, the inputs generated by kernel fuzzers need to satisfy the structural and range constraints; otherwise, it would be rejected early by input validation, thus wasting huge amounts of fuzzing time. Existing fuzzers use specifications to encode input information of system calls to address this. Based on this domain knowledge, the performance of kernel fuzzers can be improved considerably.

Syzkaller is a state-of-the-art kernel fuzzer developed by Google. In order to generate high-quality input, it utilizes the domain language Syzlang to encode system call specifications manually. Although the encoding process brings substantial costs, they enable Syzkaller to discover thousands of bugs in the Linux kernel. Meanwhile, many works improve each part of kernel fuzzing [12, 14, 21, 26, 29, 35, 38]. Take Moonshine [23] as an example, it proposes a seed distillation algorithm to collect system call sequences from real-world applications and provide initial seeds for kernel fuzzers. Healer [30] optimizes input synthesis with system call influence relations and utilizes a dynamic learning algorithm to identify such relation between calls. Although both works improve the fuzzing performance significantly, their performance, like Syzkaller, depends on the quality and abundance of Syzlang specifications.

2.2 System Call Specification

Many system calls in Linux are an abstraction over corresponding functionalities of kernel submodules, and they are responsible for dispatching the user input to submodules' operations. A system call can accept parameters with different types based on submodules' expectations. As shown in Figure 1, the input types of socket-related calls can vary for different underlying protocols. The type of parameter `val` in `setsockopt` is `void*`, where it becomes `struct tcp_repair_window` when protocol is TCP, while other protocols can define different types. The structure of `val` can be very complex since each protocol supported by the Linux kernel can utilize different types. Besides, the original type of `val` (`void*`) does not provide any structural information to fuzzer. Without further input information, fuzzer cannot test `setsockopt` effectively since most generated inputs do not satisfy the requirements of specific protocols and will be rejected by input validation.

Kernel fuzzers utilize specifications written in domain language to generate input. For instance, Syzkaller utilizes Syzlang to encode specifications for specific submodules. Within each submodule's specification, kernel experts first define the resource type corresponding to the submodule. The resource type in Syzlang infers that the value of a parameter can only be constructed by the kernel and represents a kind of kernel resource. Then, kernel experts specialize system calls

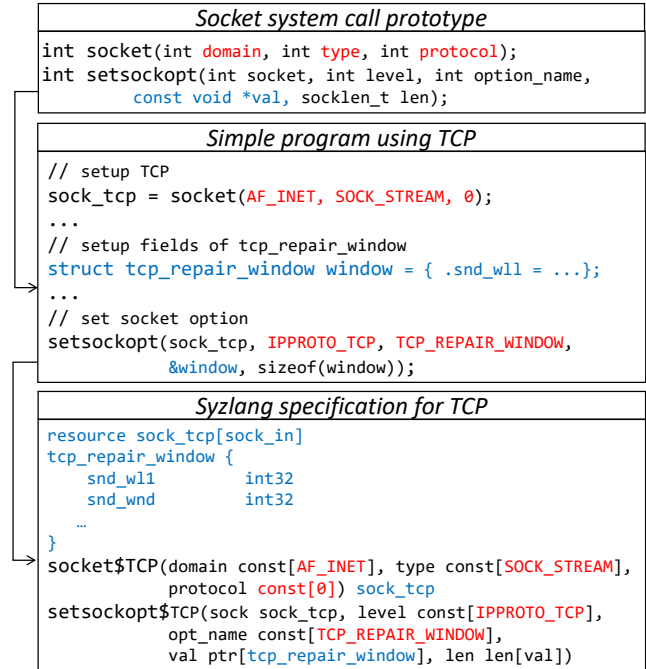


Figure 1: The input types of socket-related calls can vary for different protocols. The type of parameter `val` is `void*`, it becomes `struct tcp_repair_window` when the protocol is TCP, and other protocols can define different types. With Syzlang (the bottom part), calls can be specialized to specific protocol with range constraints (highlighted in red) and precise types (highlighted in blue).

that can access the submodule to multiple simplified calls via adding range constraints and qualifying the input type. Take Figure 1 as an example, it demonstrates parts of specifications written for TCP. The resource type `sock_tcp` represents a created TCP socket. Each parameter of the specialized call `socket$TCP` is qualified as a constant value (highlighted in red), which guides kernel fuzzers to set up TCP socket correctly. The parameter `val` of `setsockopt$TCP` is qualified as `tcp_repair_window`, which is the correct type corresponding to `TCP_REPAIR_WINDOW` option. Using Syzlang, these socket-related calls can be specialized to specific protocols with range constraints and precise types. Kernel fuzzers can use specialized calls to considerably improve their efficiency.

However, manually encoding specifications can be time-consuming due to the required domain knowledge mentioned in Section 1. Several works propose to generate specifications for specific system calls or particular scenarios. DIFUZE [6] is dedicated to generating specifications for system call `ioctl` of Android drivers and is the most relevant work to ours. It first finds all uses of `file_operations` related structures to identify the handle of `ioctl`. DIFUZE then tries to extract the device name from specific registration functions in the kernel. Finally, it detects the command values and correspond-

ing parameter structures with LLVM’s analysis capabilities, e.g., range analysis. With the above steps, DIFUZE can generate correct usages of `ioctl` from different drivers. However, most submodules’ operations are registered dynamically with unpredictable manners, which results in false negatives in DIFUZE. Besides, its pattern-based method can only be used to analyze `ioctl`. Meanwhile, Syzgen [4] and IMF [11] propose to generate specifications for close-sourced components of macOS. They capture system calls issued by userspace applications and generate specifications by analyzing the parameters’ value of captured calls. Nevertheless, both approaches do not utilize the available source code in open source scenarios to generate more effective system call specifications.

3 Challenges

3.1 Extracting Entries of Submodules

In order to generate specifications for specific submodules, we need to analyze the submodules’ entries that are invoked by system calls. Specifically, Linux defines the operations that submodules should implement with structures containing function pointers, e.g., `file_operations` and `proto_ops` as shown in Figure 2. Submodules implement these operations and register them to the kernel. We refer to these operations as entries. The responsibility of many system calls is to dispatch the input to the registered operations via indirect function call; thus, they do not contain much input information of the specific submodules. The submodules’ entries define the input types to the system calls for accessing themselves. Therefore, we need to analyze specific entries to obtain concrete input types to generate high-quality specifications. Take Figure 1 as an example, `val`’s type is `void*` and the system call `setsockopt` does not make any restriction on its concrete type. In TCP scenarios, `setsockopt` passes `val` to the entry `tcp_setsockopt`, which requires `val`’s type should be `struct tcp_repair_window*` when the option is `TCP_REPAIR_WINDOW`. We need to analyze the entry `tcp_setsockopt` to generate specifications for `setsockopt` in TCP scenarios.

However, the submodules’ entries can be registered dynamically in many situations, making it difficult to extract them. Submodules implement operations and store the function pointers in the corresponding structures, which are registered to the kernel with kinds of registration functions. The process mentioned above occurs at various times, such as kernel initialization and module loading. However, identifying the pointer’s target is challenging with current static analysis approach. The various registration points further increase the engineering efforts. For example, Figure 2 shows the definitions of `file_operations` and `proto_ops`, which contain the operations for kinds of files and sockets. Device drivers can implement `file_operations` according to their needs and register the structure to VFS during module loading.

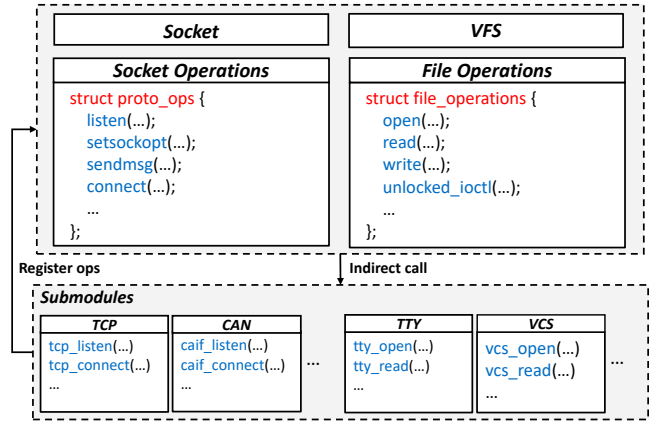


Figure 2: The definitions of `file_operations` and `proto_ops` contain the operations for files and sockets. Device drivers can implement `file_operations` as their needs and register this structure to VFS. Different protocols can register their own operations to socket layer in different ways.

Different protocols can register their `proto_ops` operations to the socket layer in different ways during kernel initializing. In order to generate specifications for submodules, we need to extract their entries and address the aforementioned dynamism.

3.2 Identifying Input Types of Entries

The second challenge is that each entry’s input type can be different across execution paths, which further increases the complexity of the analysis. As mentioned above, the kernel defines the operations that submodules need to implement via various structures containing function pointers. The number of such function pointers in each specific structure is limited, while each submodule can be very complex. Consequently, many submodules’ entries accept different input types in different execution paths to satisfy their functional requirements. In other words, the input type to the submodule’s entry is not fixed; some of the input parameters are responsible for controlling the execution path, while other parameters or fields have different types depending on the value of the former. Figure 3 shows the TCP submodule’s implementation of system call `setsockopt`, which is registered with `proto_ops` structure. The value of the parameter `optname` is mainly used to determine the different execution paths, while `optval` has different types based on the value of the former. In order to generate specifications, we need to identify the parameters’ type and collect corresponding range constraints in each execution path of the entries.

However, variables can be aliased with each other and cast to different types by different means, resulting in the difficulty in identifying their types and collecting corresponding range constraints in each execution path. To demonstrate the former


```

static int do_tcp_setsockopt(struct sock *sk, int level,
                           int optname, sockptr_t optval, unsigned int optlen)
{
    struct tcp_sock *tp = tcp_sk(sk);
    ...
    switch (optname) {
        case TCP_CONGESTION: {
            char name[TCP_CA_NAME_MAX];
            Path1: ➔ // type of `optval` is char[TCP_CA_NAME_MAX]
                    strncpy_from_sockptr(name, optval, ...);
        }
        case TCP_MAXSEG:
            int val;
            Path2: ➔ // type of `optval` is int*
                    copy_from_sockptr(&val, optval, sizeof(val));
                    tp->rx_opt.user_mss = val;
        case TCP_REPAIR_WINDOW:
            struct tcp_repair_window opt;
            Path3: ➔ // type of `optval` is tcp_repair_window*
                    if (copy_from_sockptr(&opt, optval, sizeof(opt)))
                        return -EFAULT;
    }
    return err;
}

```

Figure 3: `do_tcp_setsockopt` is TCP’s implementation of `proto_ops`. The type of parameter `optval` varies for different value of `optname`. This demonstrates that the input type of submodule’s entry can be different across execution paths

case, the value of variable `p0` with scalar type can be assigned to another variable `p1`. Meanwhile, `p1` can be cast to a pointer, which infers that `p0` represents an address. We will miss this kind of information without handling the alias between variables. To demonstrate the latter case, as shown in Figure 3, although parameter `optval` is declared as `sockptr_t` type, it is converted to different types under different cases of switch statement using different cast methods. For instance, `optval` is cast to `void*` type with C-style cast expression before the switch statement. `optval` is treated as `int*` type on *Path 2*, because `copy_from_sockptr` calls `copy_from_user` to copy `sizeof(val)` bytes from userspace, while variable `val` is `int` type. The above pattern is common in the kernel, thus we need to adequately handle the aliasing issue and type casting to properly collect types and ranges constraints.

4 Key Techniques

Figure 4 shows the overall workflow of KSG. First, the kernel source code is compiled based on the given configuration, which outputs a bootable kernel image and a series of files containing the Clang AST. The AST provides the kernel with code information for each stage of the analysis. When the kernel boots, the entry extraction module hooks multiple probes dynamically before and after specific kernel functions. KSG then scans various device files and network protocols, thus triggering the execution of hooked kernel functions, which can be captured by the probes. Consequently, the probes can detect and extract the submodules’ entries. Based on the AST and entries, KSG analyzes the range constraints and input types in

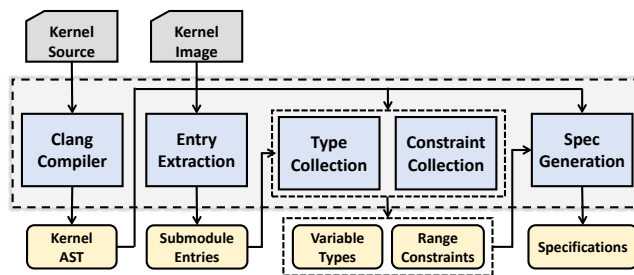


Figure 4: Workflow of KSG. The kernel code is compiled to a bootable image and files containing the clang AST. Sub-modules’ entries can be detected and extracted by the entry extraction module. Based on the AST and entries, KSG collects the range constraints and input types in each execution path of each entry with path-sensitive analysis. Finally, KSG generates specifications based on the collected information.

each execution path of each entry with path-sensitive analysis. Finally, based on the collected information, KSG generates specifications in domain language Syzlang for fuzzers, where the syntax mapping and semantics encoding are performed. The specifications can be generated with the aforementioned process, and the effectiveness of fuzzers can be improved with the generated specifications.

4.1 Entry Extraction

As mentioned above, we need to analyze the submodules’ entries for specification generation. However, the entries can be registered in many scenarios, causing difficulties in locating them. To address this, KSG utilizes a probe-based tracing to extract the entries. Although entries of different submodules can be registered with unpredictable manners, they are eventually stored into the specific data structures’ fields in the kernel. For instance, entry `file_operations` is stored into the `f_ops` field of `struct file`, which is maintained by virtual file system (VFS). In another instance, different protocols of the net subsystem store entry `proto_ops` into field `ops` of `struct socket`. Therefore, instead of analyzing the entries’ registration points, KSG extracts entries by capturing data structures containing the respective submodules’ entries, which we refer to as target structures.

Figure 5 shows the workflow of entry extraction. When the kernel boots, KSG hooks multiple probes before and after specific kernel functions utilizing Linux *eBPF* and *kprobe* (1). *eBPF* and *kprobe* enable KSG to hook our custom functions into any kernel function. We refer to these extended functions as probes and to hooked kernel functions as target functions. The target functions we choose are a mandatory part to access the submodules and are responsible for constructing target structures, thus they enable the probes to capture the execution of them and access target structures. Then, KSG scans the kernel resources corresponding to submodules from userspace.

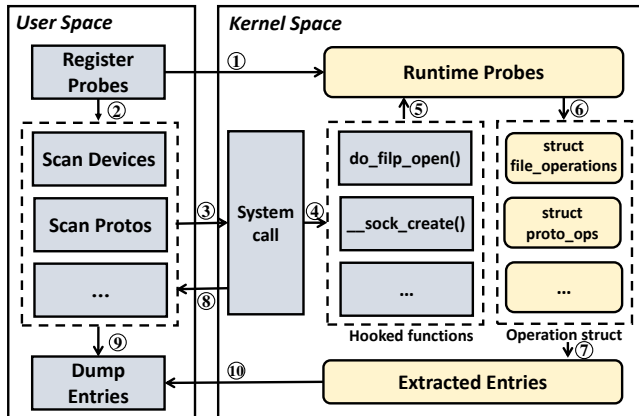


Figure 5: Workflow of entry extraction. KSG hooks probes before and after the target kernel functions. Then, it scans resources and traps into kernel space. The target functions are executed, and the whole process is captured by the probes, which extract submodules' entries and save their addresses. Finally, KSG symbolizes addresses of the entries in userspace.

For instance, KSG opens device files and specific network protocols via system calls `open` and `socket` to access `VFS` and `net` submodules (2 & 3). After trapping into kernel space, these system calls invoke the target functions, which construct the target structures (4). Then the whole process is captured by the probes, which extract the submodules' entries via accessing certain field of the target structures (5 & 6). The entries are then stored into data structures provided by eBPF so that the extracted entries can be accessed from userspace (7 & 8). Finally, KSG reads the entries via `bpf` system call, and symbolize entries to the corresponding kernel symbols with Linux `/proc/kallsyms`. The above process enables KSG to extract submodules' entries accurately.

Take device drivers for instance, the probes will be hooked after the execution of kernel function `do_filp_open`, which is called by system call `open` and is responsible for opening files used by `VFS`. Then, KSG accesses files in specific directories of the system recursively, e.g., `/dev` and `/proc`. After capturing the execution of `do_filp_open`, the probe first filters the threads, thus ensuring that only KSG's execution is captured. The probe accesses the kernel data `struct file`, which represents the state of an opened file, and reads the field `f_ops` of it. Field `f_ops` is `filer_operations` type and contains the submodule's entries. KSG saves `f_ops` into eBPF maps so that entries can be read and symbolized in userspace. Since Linux treats almost everything as a file and most submodules are accessible from `VFS`, the above procedure can extract most submodules' entries. For sockets, KSG scans all the protocols supported by the kernel and captures the kernel function `__sock_create`, which is called by system call `socket`. KSG extracts the entry `proto_ops` of each protocol by accessing the field `ops` in the `struct socket`.

4.2 Types and Constraints Collection

With submodules' entries being extracted, KSG needs to collect input information from them for specification generation. However, the parameters' types of each entry can vary across execution paths. To identify parameters' types of each execution path, KSG needs to check if a parameter, originally declared in scalar type, is cast to pointer, and collect the most precise type of each pointer. Overall, KSG utilizes the symbolic execution of Clang Static Analyzer (CSA) to perform intra-procedural, path-sensitive analysis on submodules' entries. During symbolic execution, KSG checks all expressions that can determine the parameters' types of each execution path and associate the most precise type with each parameter.

To correctly identify parameters' types, KSG first needs to handle the alias between variables. CSA associates variables with unique symbolic values and allocates a memory region for each variable based on its memory model [39]. Aliasing issue can be handled with this mechanism because CSA guarantees that variables that are aliased with each other either have the same symbol or point to the same memory region during symbolic execution. Specifically, if the symbolic value of a variable is `sym0`, then the symbolic value of variables that are assigned with the former will also be `sym0`. Variables with pointer type that have the same address during concrete execution always point to the same memory region during symbolic execution. CSA itself associates the gathered range constraints to symbolic value instead of particular variables. Since symbolic value is associated with variables and is updated accordingly during symbolic execution, range constraints can be collected and propagated by CSA.

Based on the mechanism mentioned above, we associate the type information with symbols and memory regions to collect and propagate them properly. Specifically, for variables that are originally declared in scalar type but are cast to pointers, KSG maps the symbolic value of these variables to the memory regions of pointers in `SymRegionMap` (Line 1) as shown in Algorithm 1. For pointers, KSG associates the most precise type that is known with the best effort in specific program point with their memory regions, which is stored in `RegionTypeMap` (Line 2). A special map `RegionMap` (Line 3) is used to record the connections between regions in a pointer to pointer cast. `RegionMap` is needed because CSA creates new element region for this kind of cast, while these regions represent the same variable semantically. The above three global maps can be used to record and propagate the collected type information during symbolic execution.

KSG collects input types in each execution path during CSA's symbolic execution procedure. Specifically, whenever CSA executes the type cast expression, including explicit C-style casts and implicit casts, and kernel functions with copy

Algorithm 1: Collecting Types

```
1 SymRegionMap := ∅
2 RegionTypeMap := ∅
3 RegionMap := ∅
4 for CastExpr ∈ Entry do
5   S := SourceSym(CastExpr)
6   T := TargetSym(CastExpr)
7   if IsIntegerToPtr(CastExpr) then
8     R := Region(T)
9     SymRegionMap[S] := R
10    continue
11  if !IsPtrToPtr(CastExpr) then
12    continue
13  R0 := Region(S)
14  R1 := Region(T)
15  Record(R0, R1, RegionMap)
16  STy := KnownType(R0, RegionTypeMap)
17  TTy := KnownType(R1, RegionTypeMap)
18  if IsMorePrecise(STy, TTy) then
19    updateRegionType(R1, STy)
20  else
21    updateRegionType(R0, TTy)
```

semantics, such as `copy_from_user`, KSG obtains the type information from the expressions and updates the global maps mentioned above based on the comparison rules shown in Table 1. As shown in Algorithm 1, KSG records the mapping between the symbolic value of the scalar and the memory region of the pointer, which handles the integer to pointer casts (Lines 8 to 10). The recorded mapping can be used to retrieve the region of a pointer that is declared in scalar type. For a pointer to pointer cast, the algorithm first gets the respective regions of the source pointer and the target pointer (Lines 13 to 14), and records the connection between these regions into `RegionMap` (Line 15). Then it identifies the current known type of regions with `RegionTypeMap`, and the declared type of region is used if it has not been recorded (Lines 16 to 17). Based on the rules in Table 1, the algorithm updates the regions with the more precise type (Lines 18 to 21). For kernel functions with copy semantics, KSG utilizes a similar approach. Take `copy_from_user` as an example, KSG gets the current known type of the source pointer and the target pointer, and performs the type comparison first. Then, it checks if the last parameter is a unary expression `sizeof`, if so, KSG performs an additional comparison with the corresponding type. KSG updates the `RegionTypeMap` with the most precise type. Furthermore, KSG records the data flow direction of pointers based on the analyzed kernel functions. For example, `copy_from_user` infers the `In` direction and KSG associates this information with the corresponding memory region.

To associate the memory region with the most precise type in each execution path, KSG utilizes type comparisons. As shown in Table 1, the algorithm divides the types into four categories. First, `void` or `void*` is less precise than all other types because they do not encode any structural information. Scalar type that has longer bit width is more precise than another scalar type. Both scalar and compound type are less precise than pointer type, because it’s a common use case in kernel to store the pointer value to scalar or pointer-sized compound type. For pointer types, the algorithm applies the above rules to the underlying type recursively. With the procedure above, KSG can identify the concrete type of each parameter and field of compound type.

Table 1: Rules for comparison between source type and target type. ‘>’ represents that the source type is more precise than the target type, ‘<’ is the opposite. *Size* means that the result depends on the size of comparison types. *Underlying* means comparing the underlying type recursively.

	Void	Scalar	Compound	Ptr
Void	=	<	<	<
Scalar	>	<i>Size</i>	<	<
Compound	>	>	<i>Size</i>	<
Ptr	>	>	>	<i>Underlying</i>

Figure 6 shows a running example of `do_tcp_setsockopt` with Algorithm 1. First, CSA marks the input parameter `optname` and `optval` as symbol `sym0` and `sym1`, respectively. After the first case condition, it collects range constraint of `sym0`, indicating that `optname` equals to `TCP_REPAIR_OPTIONS` on the current execution path. Algorithm 1 is invoked when CSA enters the kernel function `copy_from_sockptr` since it calls `copy_from_user` eventually. Symbolic value `sym1` is associated with memory region `region0`, which is recorded in `SymRegionMap`, because `optval` (integer type) is cast to a pointer. The mapping from `region0` to `struct tcp_repair_opt` is also recorded in `RegionTypeMap`. Finally, CSA captures another range constraint of `opt`’s field `opt_code`. In this way, KSG knows the concrete type of `sym1` based on the information in `SymRegionMap` and `RegionTypeMap`. This example demonstrates that the types and range constraints can be collected properly and the second challenge can be addressed by combining CSA and type collection.

4.3 Specification Generation

Leveraging the above approach, KSG can collect parameters’ types and range constraints in each execution path of submodules’ entries. Based on the collected information, KSG generates specifications in domain language `Syzlang` for kernel fuzzers. The generation procedure needs to accomplish two

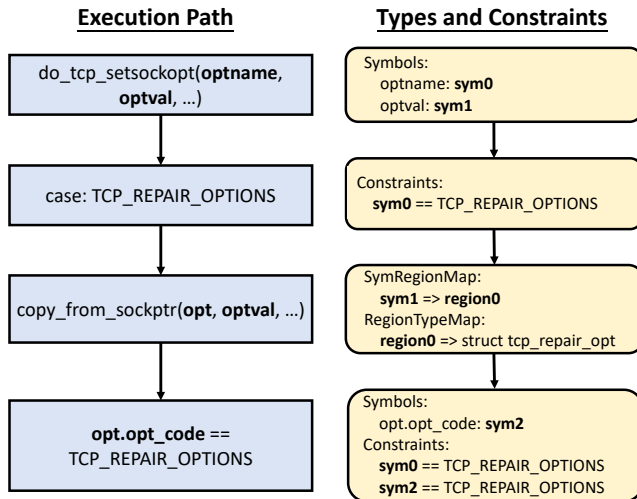


Figure 6: Running example for Algorithm 1. CSA first marks `optname` and `optval` as symbolic value `sym0` and `sym1`. Then it captures range constraint on symbolic value `sym0`. Algorithm 1 maps `sym1` to memory region `region0` since `optval` is cast to pointer type. Finally, CSA further captures another range constraint of `sym2`, symbolic value of `opt`'s field `opt_code`.

major goals: syntax mapping and semantic encoding. The former performs the mapping from C language AST to Syzlang AST and the latter encodes the collected range constraints into the generated specifications.

KSG divides the generation process into two steps. The first step generates the definitions of Syzlang resource type corresponding to the submodule, and the system calls that are responsible for creating the former. As mentioned in Section 4.1, KSG scans device files and protocols to extract submodules' entries. Meanwhile, the needed information for accessing the submodule is recorded. For instance, the file paths are saved for device drivers and the `domain`, `type` and `proto` are saved for specific sockets. Based on this information, KSG defines *resource* type for each submodule, and the name of the defined *resource* type follows specific rules. For example, resource types for device and socket are prefixed with `fd` and `sock`, respectively. For TCP submodule shown in Figure 1, `sock_tcp` is defined in this step. Then, KSG generates the corresponding system calls that create the resource type. For example, KSG generates the system call `open` for device drivers, and the input path of `open` is qualified to the file path of the device. The specialized version of system call `socket` is generated for each protocol, e.g., `socket$TCP` shown in Figure 1.

The second step generates the specialized calls for the remaining entries of a submodule. Specifically, KSG generates a specialized call for each execution path of each entry. The duplicated calls are filtered, and the parameters' types of each generated specialized call are qualified to the corresponding type in the execution path. Specifically, for a variable de-

clared in scalar type, KSG first checks if it is a pointer via querying `SymRegionMap` and maps it to Syzlang pointer if so; otherwise, KSG obtains its bit size according to the AST information and maps it to the corresponding numeric type with same bit size in Syzlang. Meanwhile, KSG checks whether the symbol of the scalar has range constraints by querying the program state of CSA. The corresponding constraint is represented as Syzlang's `const` type or ranged integer type. For array type, KSG first maps its element type to Syzlang type recursively, then queries CSA whether its length has range constraints. KSG constructs the corresponding Syzlang array type based on the mapped element type and length information. For pointer type, KSG first gets the memory region of the pointer from CSA, and queries the concrete type associated with the region from `RegionTypeMap`. KSG then maps the type of pointee recursively, and queries data flow direction associated with the memory region. KSG constructs the Syzlang pointer type with the mapped pointee's type and collected data flow direction. Finally, KSG maps each field of compound type to Syzlang type and generates the corresponding compound type in Syzlang. Based on the above mapping rules, KSG can generate specifications for submodules' entries based on the collected types and range constraints.

Take Figure 1 as an example, KSG generates three specialized system calls for each path of `do_tcp_setsockopt`. The type of `optname` is mapped to `const` type in Syzlang based on range constraints of each path. In the meantime, the type of `optval` is mapped to `array`, `int32`, and `struct`, respectively. Listing 1 in the Appendix shows part of generated specifications for driver `/dev/pts`, which manual specifications do not cover. Listing 2 shows part of generated specifications for the socket `X25`.

5 Implementation

Entry extraction. We implement eBPF programs based on BCC [10] and hook them as kprobe into target kernel functions. Two probes are used to capture the entries of device drivers' and protocols' operations, respectively. We currently utilize a userspace program to trigger the extraction process. The program first attaches the probes to the kernel. It then scans kernel-provided resources, such as opening files in `/dev`, mounting all the supported file systems, opening files in different file systems, and creating all the supported sockets of the kernel. These operations allow us to extract the implementation of the corresponding file operations and socket operations for different submodules.

Types and Constraints. We implement the types and constraints collection based on Clang13. Algorithm 1 is implemented as multiple CSA checkers that are hooked after each time CSA simulates execution of cast expression and before the execution of functions with copy semantics, e.g., `copy_from_user`. These checkers read the symbol values and memory regions of the expressions in the hooked opera-

tions from current program state, and update the type information stored in the global maps based on the type comparison rules in Table 1. For better intra-procedure analysis, we utilize the cross translation unit (CTU) analysis of CSA based on pre-dumped AST and compilation database. We customized the analysis configuration, e.g., increasing the max number of imported translation units, limiting the loop time since it does not provide new information for specification generation but reduces efficiency. Besides, we also modeled a larger number of kernel library APIs via implementing CSA checkers for better symbolic execution, including `kmalloc`, string manipulation functions, etc. These checkers observe the symbolic execution of the kernel and actively participate in modeling the program behavior via modifying the region bindings and range constraints stored in the program state.

Specification Generation. The generation procedure is implemented as plugins too that are hooked into CSA at the end of each execution path’s simulation. We first implement AST to fully support Syzlang language. Based on the type information stored in the global maps and the range constraints of each symbol in the CSA, the translation of KSG maps the C language AST to Syzlang AST. In order to generate the specifications, the mapped AST is serialized into text format that conforms to the syntax rules of Syzlang, thus allowing kernel fuzzers to use the generated specifications and speed up the entire fuzzing campaign.

6 Evaluation

In this section, we evaluate the effectiveness of KSG on recent versions of Linux and fuzzers. Specifically, we chose Linux-5.15, 5.10, and 5.4 as our target versions. Linux 5.15 is the latest version prior to submission, whereas 5.10 and 5.4 are widely used by many distributions. To evaluate the effectiveness of the generated specifications in improving fuzzers’ performance, we took the generated specifications as input to Syzkaller and Moonshine, and compared the code coverage and bug finding capabilities to their original version. We chose Syzkaller because it is the state-of-the-art kernel fuzzer. Moonshine improves Syzkaller’s fuzzing efficiency by distilling high-quality seeds for it and is a representative fuzzer. We design experiments to address the following questions:

- **RQ1:** How does KSG perform in generating system call specifications in terms of efficiency and quality?
- **RQ2:** How effective are the generated specifications in improving the coverage of kernel fuzzers?
- **RQ3:** How effective are the generated specifications in assisting kernel fuzzers to find bugs?

Experiment Settings The experiments were conducted on a Linux server with a 16-core Intel i7-10700K CPU and

32 GiB of memory. Each version of the kernel uses the same compilation configuration. Specifically, `CONFIG_BPF` and `CONFIG_KPROBE` were enabled for entry extraction. We also enabled `CONFIG_KCOV` to collect code coverage. We extended fuzzers with the generated specifications, and we refer to those extended fuzzers as Syzkaller+ and Moonshine+, respectively. All 4 fuzzers were configured with the same parameters in terms of QEMU configurations and base system call specifications. Specifically, we started all experiments simultaneously and distributed the resources evenly, including 2 cores and 4 GiB of memory for each virtual machine. All 4 fuzzers adopted the same base version of the Syzlang specifications. To reduce statistical errors, each experiment was repeated 3 times and executed over a period of 72 hours, and the average results were reported.

6.1 Specification Generation

We executed KSG on three versions of the Linux kernel and the whole process of specification generation is automatic. Table 2 shows the results of this process. During entry extraction, KSG scanned 1098 unique device files and 78 different sockets in total, and extracted 572 and 222 entries, respectively. Note that the number of entries is not equal to the number of scanned files and sockets multiplied by the number of function pointers defined in `file_operations` and `proto_ops`. This is because: first, the registered operations of different files and sockets can be the same; second, each submodule does not need to implement all the operations; finally, KSG de-duplicates the extracted entries and verifies the extracted addresses. Besides, we manually verified the correctness of extracted entries by reading the source code corresponding to the submodule. The result shows that KSG can correctly extract the entries of all files and sockets that are accessible from userspace. Furthermore, since KSG performs entry extraction dynamically based on eBPF after kernel booted and all submodules loaded, the correctness of the extracted entries can also be guaranteed.

Table 2: KSG extracted 792 entries by scanning 78 sockets and 1098 device files. After path-sensitive analysis in 5h, KSG generated specifications containing 2433 specialized calls, and 1460 of them are new to existing specifications.

	Scanned	Entries	Specs	New Specs
Socket	78	222	923	+586
Driver	1098	572	1510	+874
Overall	1176	794	2433	+1460

By performing path-sensitive analysis on submodules’ entry, KSG generates 8 specialized calls per minute, with a total of 2433 specialized calls generated in 5 hours. Of this total, 1510 specialized calls are generated from device drivers while 923 specialized calls corresponded to sockets. Specifically, for

64% of the extracted entries, the number of generated specialized calls is less than 2. This is because the input types remain consistent across the execution paths. For instance, KSG generates one specialized call for system call `bind` of each type of socket. The input address is qualified to the type defined by the corresponding socket type, while the type of such parameter in C prototype does not constrain the input structure. Although the number of specialized calls for this 64% of the entries is limited, encoding specifications for them requires extensive domain knowledge, whereas KSG can automate this process leveraging the source code analysis. For 36% of the extracted entries, the number of generated specialized calls is more than 2 because the input types of these entries can vary in different execution paths. The average number of specialized calls for these entries is 4, and KSG generates up to 29 specialized calls for system call `getsockopt` of X25 socket. For those 36% of the entries, the manual efforts of writing specifications would be vast, while the automation of KSG can significantly reduce the time cost of this process.

Compared with the existing specifications that contains 1204 specialized calls for the drivers and sockets scanned by KSG, 1460 generated calls are new, of which 586 and 874 are generated from the analyzed sockets and drivers, respectively. In order to further verify the correctness of the generated specifications, we manually checked if the range constants match the collected parameter types by reading the source code of submodules. The final result shows that KSG can correctly extract the input types and the corresponding range constraints in each execution path of submodules' entries.

6.2 Coverage Improvement

To answer **RQ2**, we took 1460 new specialized calls as input to Syzkaller and Moonshine, while monitoring the fuzzing process and sampling each fuzzer's statistics in the 72-hour run. Figure 7 shows the comparison of branch coverage between fuzzers and Table 3 lists detailed statistics. The base specifications used by Syzkaller and Moonshine contain 4144 specialized calls in total, including specifications encoded for submodules that have not been handled by KSG. With 1460 new specialized calls, Syzkaller+ and Moonshine+ achieved 22% and 23% coverage improvement, respectively.

As shown in Figure 7, both Syzkaller+ and Moonshine+ can achieve higher coverage statistics than their original version in the same amount of time. Specifically, all tools show significant growth in the first 8 hours, where the advantage of the generated specifications is not obvious. After fuzzing for 8 hours, the coverage growth of Syzkaller and Moonshine starts to slow down, whereas that of Syzkaller+ and Moonshine+ is significantly faster than the former. This is because KSG does not improve kernel fuzzers' throughput or efficiency, but rather enables fuzzers to reach more modules and code with additional generated specifications. Therefore, all fuzzers perform at similar rates before 8 hours since they have yet

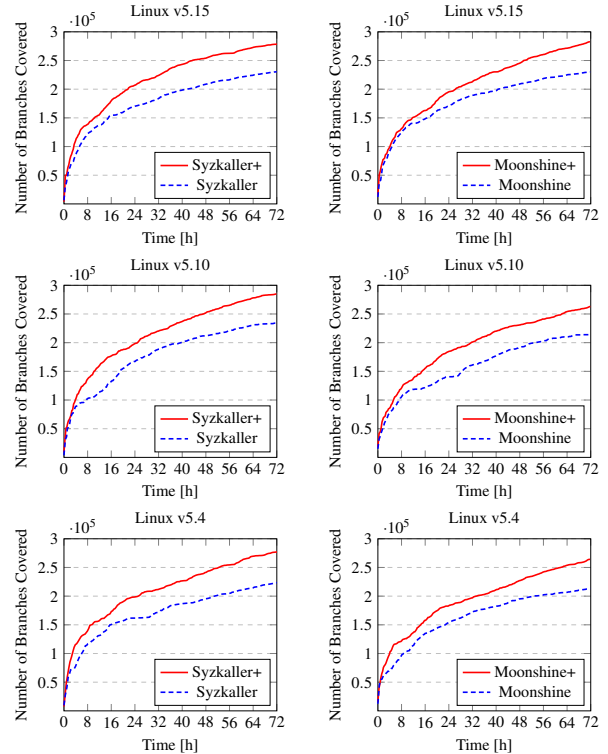


Figure 7: Coverage growth of Syzkaller and Moonshine with generated system call specifications on three versions of Linux kernel over 72 hours. In all kernel versions, Syzkaller+ and Moonshine+ achieve the higher coverage statistics.

to cover the code reachable using manually-written specifications. They diverge after 8 hours as the manually-written specifications cannot provide the kernel fuzzers with more low hanging fruit, while the generated specifications allow the fuzzers to continue finding more code in more modules.

In principle, fuzzers utilize the generated specifications to generate test cases. In order to further demonstrate the reason behind the improvement, we analyzed the output corpus of all fuzzers and calculated the percentage of test cases that contain the newly generated calls in the whole corpus. At the end of 72-hour experiment, the average percentage of such inputs in the corpus is 28%. Therefore, the reason behind the coverage improvement is that the generated specifications provide new test portals for fuzzers. Syzkaller+ and Moonshine+ can synthesize test cases based on the new specifications thus covering kernel code that used to be unreachable. Meanwhile, the improvement can demonstrate the quality of the generated specifications, since specifications with low quality, e.g., range constraints mismatch input types, can even hinder fuzzers' capabilities. The above results prove that the generated specifications can assist fuzzers in exploring more code in the kernel.

Table 3: Coverage statistics of fuzzers compare to their original versions. Columns “min-impr” and “max-impr” present the minimum / maximum improvement.

(a) Syzkaller+ vs. Syskaller			
Version	min-impr	max-impr	Average
5.15	+18%	+24%	+21%
5.10	+19%	+25%	+22%
5.4	+20%	+28%	+24%
Overall	+19%	+25%	+22%

(b) Moonshine+ vs. Moonshine			
Version	min-impr	max-impr	Average
5.15	+19%	+24%	+22%
5.10	+20%	+25%	+23%
5.4	+20%	+26%	+24%
Overall	+19%	+25%	+23%

6.3 Bug Finding and Case Studies

To answer **RQ3**, we tested the Linux kernel with Syzkaller+ and Moonshine+ for two weeks. As a result, we found 138 unique vulnerabilities in total and 26 were confirmed by maintainers as previously unknown bugs, of which 13 and 6 were fixed and assigned with CVEs, respectively. Table 4 lists the details of those vulnerabilities. Most of these vulnerabilities are critical. For instance, KSG assisted fuzzers to discover a vulnerability with a 7.0 CVSS score (CVE-2021-4028). Although Syzkaller has been testing the Linux kernel continuously with large amounts of computing resources, these 26 vulnerabilities have not been reported. The reason why KSG assisted Syzkaller+ and Moonshine+ to discover 26 previously unknown vulnerabilities is that the generated specifications provide fuzzers with more information of system calls. Based on this domain knowledge, Syzkaller+ and Moonshine+ can generate test cases to test kernel code that used to be difficult for fuzzers to reach. The above result shows that the specifications automatically generated by KSG can improve the fuzzers’ vulnerability detection capabilities.

Case Study: CVE-2021-4148 KSG assisted fuzzers to discover a vulnerability in VFS. As shown in Figure 8, `block_invalidatepage()` would throw BUG due to assertion failure if `stop` is greater than `PAGE_SIZE`. However, the input generated by the fuzzer is a huge page, and the length is the size of the huge page due to the read-only FS THP support. This triggers kernel crash directly and Figure 8 shows a fix for this. However, the root cause of this vulnerability is complicated. Specifically, the kernel isn’t supposed to get a writable file descriptor on a file that has huge pages added to the page cache without the filesystem’s knowledge. VFS should have truncated the page cache when it found THPs in the cache. Except for the fix mentioned, this vulnerabil-

Table 4: KSG assisted fuzzers to discover 26 previously unknown vulnerabilities. All of these vulnerabilities have been confirmed by maintainers; 13 of these bugs have been fixed by corresponding patches and another 6 have been assigned with CVEs.

Operation	Risk	Status
<code>sk_stream_kill_queues</code>	logic bug	Fixed
<code>__init_work</code>	use after free	CVE-2021-4150
<code>truncate_inode_page</code>	logic bug	Fixed
<code>__folio_mark_dirty</code>	logic bug	Fixed
<code>kvm_arch_vcpu_create</code>	logic bug	CVE-2021-4032
<code>cma_cancel_listens</code>	use after free	Fixed
<code>io_wq_submit_work</code>	logic bug	CVE-2021-4023
<code>btrfs_alloc_tree_block</code>	logic bug	Fixed
<code>__btrfs_tree_lock</code>	deadlock	CVE-2021-4149
<code>smp_call_function</code>	soft lockup	Confirmed
<code>block_invalidatepage</code>	dereference null	CVE-2021-4148
<code>rdma_listen</code>	use after free	CVE-2021-4028
<code>ext4_block_write_begin</code>	logic bug	Confirmed
<code>io_ring_exit_work</code>	task hung	Fixed
<code>skb_try_coalesce</code>	task hung	Confirmed
<code>btrfs_search_slot</code>	deadlock	Fixed
<code>__set_page_dirty</code>	logic bug	Confirmed
<code>__kernel_read</code>	logic bug	Fixed
<code>xlog_cil_commit</code>	dereference null	Fixed
<code>hub_port_init</code>	task hung	Confirmed
<code>hci_cmd_timeout</code>	logic bug	Confirmed
<code>cgroup_rstat_flush_locked</code>	data race	Fixed
<code>btrfs_free_tree_block</code>	logic bug	Confirmed
<code>io_uring_cancel_generic</code>	task hung	Fixed
<code>hci_uart_tx_wakeup</code>	logic bug	Fixed
<code>blk_mq_get_tag</code>	logic bug	Fixed

ity was fixed properly with additional patches. Leveraging the newly generated specifications, the fuzzer synthesized the corresponding test cases thus triggering the assertion failure.

7 Discussion and Limitations

During the experiments, we found a total of 231 specialized calls from existing specifications that are encoded for the submodules scanned by KSG, but are nonexistent in the generated specifications. We believe there are three major reasons for this. First, KSG mainly considers range constraints while handwritten specifications encode other parameters semantics, e.g., defining parameters that are checksums of other fields as instances of the `csum` type in Syzlang. Second, kernel experts can redefine original input types from system call definitions to other types with the same memory layout to generate argument values more efficiently. For instance, some submodules use the high 16 bits and low 16 bits of a `u32` number for different purposes, and kernel experts redefine them as two `u16` types so that fuzzer can generate and mutate values for them individually. For these two limitations, we can improve KSG further by hooking more kernel functions during

```

diff --git a/fs/buffer.c b/fs/buffer.c
index ab7573d72dd7..4bcb54c4d1be 100644
--- a/fs/buffer.c
+++ b/fs/buffer.c
@@ -1507,7 +1507,7 @@ void block_invalidatepage(struct
page *page, unsigned int offset,
/*
 * Check for overflow
 */
-   BUG_ON(stop > PAGE_SIZE || stop < length);
+   BUG_ON(stop > thp_size(page) || stop < length);

   head = page_buffers(page);
   bh = head;
@@ -1535,7 +1535,7 @@ void block_invalidatepage(struct
page *page, unsigned int offset,
 * The get_block cached value has been
unconditionally invalidated,
 * so real IO is not possible anymore.
 */
-   if (length == PAGE_SIZE)
+   if (length >= PAGE_SIZE)
       try_to_release_page(page, 0);
out:
   return;

```

Figure 8: When the size of the `stop` is greater than `PAGE_SIZE`, `block_invalidatepage()` would throw BUG. Fuzzer triggered this crash by passing a huge page, where the length is the size of huge page due to FS THP support. This figure shows a direct fixing patch for CVE-2021-4148.

path-sensitive analysis to collect more parameters’ semantics as well as redefining input types based parameters’ usage. Finally, the kernel code contains low-level operations, e.g., inline assembly, which may not be well modeled by CSA, thus leading to the range constraints and type information being missed during the analysis procedure. To address this, we can construct checkers to simulate common low-level operations so that the related information can be collected and propagated properly.

Currently, we mainly apply KSG to generate specifications for drivers and sockets. Since many resources in Linux are represented as files in VFS, using file-operation-relevant system calls allows us to extract entry information for many submodules. Meanwhile, in principle, KSG is generalizable. For other multiplexing system calls, we can adapt entry extraction to the target through a slight analysis of the internal implementation to find the kernel functions that need to be injected; then, we can apply the rest of KSG. For other system calls, we can directly execute the collecting algorithm of KSG and generate specifications based on gathered information since these steps only depend on the source code information. Take system call `prctl()` as an example, KSG can collect the argument constraints directly from `sys_prctl()`.

8 Conclusion

In this paper, we propose KSG to automatically generate system call specifications for kernel fuzzers based on entry

extraction and types and constraints collection. The evaluation shows that KSG generates 8 specialized calls per minute, with a total of 2433 specialized calls generated in 5 hours. Leveraging the generated specifications, Syzkaller and Moonshine’s coverage were improved 22% and 23%, respectively. Furthermore, KSG assisted fuzzers to discover 26 previously unknown bugs. The above result demonstrates that KSG is effective in generating system call specifications, and the generated specifications can improve the fuzzers’ performance.

For future work, we will extend KSG to other system calls or submodules to generate more specifications since some submodules are not covered by KSG yet, and submodules like drivers can change over time, which potentially involves making modifications to entry extraction. More importantly, we can augment KSG to infer semantic information of system calls’ parameters, thus significantly improving the generated specifications, which can be implemented with multiple CSA checkers encoded carefully with domain knowledge.

Acknowledgments

We sincerely appreciate the guidance from our shepherd. We would also like to thank the anonymous reviewers for their valuable comments and input to improve our paper. This research is sponsored in part by the NSFC Program (No. 62022046, 92167101, U1911401, 62021002, 62192730), National Key Research and Development Project (No. 2019YFB1706200, No2021QY0604).

References

- [1] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. FUDGE: Fuzz Driver Generation at Scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019*, page 975–985, New York, NY, USA, 2019. Association for Computing Machinery.
- [2] Daniel Borkmann. Linux eBPF. <https://ebpf.io>.
- [3] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.
- [4] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. SyzGen: Automated Generation of Syscall Specification of Closed-Source MacOS Drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS ’21*, page 749–763, New York, NY, USA, 2021. Association for Computing Machinery.

- [5] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1967–1983, Santa Clara, CA, August 2019. USENIX Association.
- [6] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, page 2123–2138, New York, NY, USA, 2017. Association for Computing Machinery.
- [7] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-Based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08*, page 206–215, New York, NY, USA, 2008. Association for Computing Machinery.
- [8] Google. Kernel address sanitizer. <https://www.kernel.org/doc/html/latest/dev-tools/kasan.html>.
- [9] Google. Kernel concurrency sanitizer. <https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html>.
- [10] Brendan Gregg. BPF Compiler Collection. <https://www.iovisor.org/technology/bcc>.
- [11] HyungSeok Han and Sang Kil Cha. IMF: Inferred Model-Based Fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, page 2345–2358, New York, NY, USA, 2017. Association for Computing Machinery.
- [12] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Ruzzer: Finding Kernel Race Bugs through Fuzzing. In *IEEE Symposium on Security and Privacy*, pages 754–768. IEEE, 2019.
- [13] Jim Keniston. Linux Kprobe. <https://www.kernel.org/doc/html/latest/trace/kprobes.html>.
- [14] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *NDSS*, 2020.
- [15] lcamtuf. American fuzzy lop, 2013. <https://lcamtuf.coredump.cx/afl/>.
- [16] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, page 475–485, New York, NY, USA, 2018. Association for Computing Machinery.
- [17] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 154–170, Los Alamitos, CA, USA, may 2022. IEEE Computer Society.
- [18] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. PAFL: Extend Fuzzing Optimizations of Single Mode to Industrial Parallel Mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018*, page 809–814, New York, NY, USA, 2018. Association for Computing Machinery.
- [19] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 562–566, 2018.
- [20] LLVM Developer Group. Clang Static Analyzer. <https://clang-analyzer.llvm.org/>.
- [21] Dominik Maier, Benedikt Radtke, and Bastian Harren. Unicorefuzz: On the Viability of Emulation for Kernel-space Fuzzing. In *Proceedings of the 13th USENIX Conference on Offensive Technologies, WOOT'19*, page 8, USA, 2019. USENIX Association.
- [22] Andy Nguyen. CVE-2020-12352, 2020. <https://nvd.nist.gov/vuln/detail/CVE-2020-12352>.
- [23] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, Baltimore, MD, August 2018. USENIX Association.
- [24] Manfred Paul. CVE-2021-3490, 2021. <https://nvd.nist.gov/vuln/detail/CVE-2021-3490>.
- [25] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing Seed Selection for Fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 861–875, San Diego, CA, August 2014. USENIX Association.
- [26] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, Vancouver, BC, August 2017. USENIX Association.

- [27] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.
- [28] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, page 62–71, New York, NY, USA, 2009. Association for Computing Machinery.
- [29] Yuheng Shen, Hao Sun, Yu Jiang, Heyuan Shi, Yixiao Yang, and Wanli Chang. Rtkaller: State-Aware Task Generation for RTOS Fuzzing. *ACM Trans. Embed. Comput. Syst.*, 20(5s), sep 2021.
- [30] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. *HEALER: Relation Learning Guided Kernel Fuzzing*, page 344–358. Association for Computing Machinery, New York, NY, USA, 2021.
- [31] Dmitry Vyukov and Andrey Konovalov. Syzbot, 2015. <https://syzkaller.appspot.com/upstream>.
- [32] Dmitry Vyukov and Andrey Konovalov. Syzbot Dashboard, 2015. <https://storage.googleapis.com/syzkaller/cover/ci-qemu-upstream.html>.
- [33] Dmitry Vyukov and Andrey Konovalov. Syzkaller: an unsupervised coverage-guided kernel fuzzer, 2015. <https://github.com/google/syzkaller>.
- [34] Dmitry Vyukov and Andrey Konovalov. Syzlang: System Call Description Language, 2015. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.
- [35] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2741–2758. USENIX Association, August 2021.
- [36] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, ICSE '18, page 61–64, New York, NY, USA, 2018. Association for Computing Machinery.
- [37] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 147–159. USENIX Association, July 2021.
- [38] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660, 2020.
- [39] Zhongxing Xu, Ted Kremenek, and Jian Zhang. A memory model for static analysis of c programs. In *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I*, ISOFA'10, page 535–548, Berlin, Heidelberg, 2010. Springer-Verlag.
- [40] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. Intelligen: automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 318–327. IEEE, 2021.

9 APPENDIX

9.1 Generated Specifications

Listing 1 shows part of generated system call specifications for `/dev/pts`. During entry extraction, KSG extracts `struct file_operation` of device `/dev/pts` via accessing device files under `/dev/pts` directory. KSG then performs path sensitive analysis on each extracted entry to collect types and range constraints. During the first step of generation, KSG defines the resource type `fd_dev_pts_0`, where the name of resource type is generated based the accessed file. `openat$dev_pts_0_0` is also generated during this step and the target file path is qualified to `/dev/pts/0`. After the second step of the generation, the rest of specialized calls and related types were generated based on the collected types and range constraints.

Listing 1: Generated specifications for `/dev/pts` driver

```
resource fd_dev_pts_0[fd]
openat$dev_pts_0_0(fd const[AT_FDCWD], file
  ptr[in, string["/dev/pts/0"]], flags flags
  [open_flags], mode flags[open_mode])
fd_dev_pts_0
...
ioctl$dev_pts_0_16(fd fd_dev_pts_0, cmd const
  [0x5413], arg ptr[in, winsize])
ioctl$dev_pts_0_11(fd fd_dev_pts_0, cmd const
  [0x80045440], arg ptr[in, int32])
ioctl$dev_pts_0_3(fd fd_dev_pts_0, cmd const[0
  x541f], arg ptr[in, serial_struct])
ioctl$dev_pts_0_5(fd fd_dev_pts_0, cmd const[0
  x545d], arg ptr[in, serial_icounter_struct
  ])
...

serial_icounter_struct {
  cts int32
  dsr int32
  rng int32
  dcd int32
  rx int32
  tx int32
  frame int32
  overrun int32
  parity int32
  brk int32
  buf_overrun int32
  reserved array[int32, 9]
}
serial_struct {
  ...
  closing_wait2 int16
  iomem_base ptr[out, array[int8]]
  ...
}
winsize {
  ws_row int16
  ws_col int16
  ws_xpixel int16
  ws_ypixel int16
}
```

Listing 2 shows part of generated system call specifications for socket X25. During entry extraction, KSG extracts `struct proto_ops` of X25 via invoking system call `socket` with address family `AF_X25`. KSG then performs path sensitive analysis on each extracted entry to collect types and range constraints. During the first step of generation, KSG defines the resource type `sock_X25_SeqPacket`, where the name of resource type is generated based the address family and socket type. `socket$X25_SeqPacket` is also generated during this step and the parameters are qualified to corresponding constant. After the second step of the generation, the rest of specialized calls and related types were generated based on the collected types and range constraints.

Listing 2: Generated specifications for X25 socket

```
resource sock_X25_SeqPacket[sock]

socket$X25_SeqPacket(domain const[0x9], type
  const[0x5], proto const[0x0])
sock_X25_SeqPacket
bind$X25_SeqPacket_0(sock sock_X25_SeqPacket,
  addr ptr[in, sockaddr_x25], len bytesize[
  addr])
...
setsockopt$X25_SeqPacket_0(sock
  sock_X25_SeqPacket, level const[0x106],
  opt_name const[0x1], buf ptr[in, int32],
  len ptr[in, int32])
...
ioctl$X25_SeqPacket_6(fd sock_X25_SeqPacket,
  cmd const[0x89e5], arg ptr[in,
  x25_calluserdata])
ioctl$X25_SeqPacket_4(fd sock_X25_SeqPacket,
  cmd const[0x89ec], arg ptr[in,
  x25_causeddiag])
ioctl$X25_SeqPacket_9(fd sock_X25_SeqPacket,
  cmd const[0x89ea], arg ptr[in,
  x25_dte_facilities])
ioctl$X25_SeqPacket_10(fd sock_X25_SeqPacket,
  cmd const[0x89e3], arg ptr[in,
  x25_facilities])
...
sockaddr_x25{
  sx25_family const[0x9, int16]
  sx25_addr x25_address
}
x25_address{
  x25_addr array[int8, 16]
}
x25_calluserdata {
  cudlength int32
  cuddata array[int8, 128]
}
x25_causeddiag {
  cause int8
  diagnostic int8
}
x25_dte_facilities {
  ...
}
x25_facilities {
  ...
}
```


DLOS: Effective Static Detection of Deadlocks in OS Kernels

Jia-Ju Bai
Tsinghua University

Tuo Li
Tsinghua University

Shi-Min Hu
Tsinghua University

Abstract

Deadlocks in OS kernels can cause critical problems like performance degradation and system hangs. However, detecting deadlocks in OS kernels is quite challenging, due to high complexity of concurrent execution and large code bases of OS kernels. In this paper, we design a practical static analysis approach named DLOS, to effectively detect deadlocks in OS kernels. DLOS consists of three key techniques: (1) a *summary-based lock-usage analysis* to efficiently extract the code paths containing distinct locking constraints from kernel code; (2) a *reachability-based comparison method* to efficiently detect locking cycles from locking constraints; (3) a *two-dimensional filtering strategy* to effectively drop false positives by validating code-path feasibility and concurrency. We have evaluated DLOS on Linux 5.10, and find 54 real deadlocks, with a false positive rate of 17%. We have reported these deadlocks to Linux kernel developers, and 31 of them have been confirmed.

1 Introduction

Concurrent execution improves the performance of OS kernels, but can inevitably introduce concurrency bugs. Some studies [37, 38, 51] have shown that a large part of reported OS bugs are related to kernel concurrency. Deadlock is a common kind of concurrency bugs, caused by a locking cycle in different threads. For example, one thread acquires the locks A and then B , while the other concurrent thread acquires the locks B and then A , and thus a deadlock caused by the ABBA locking cycle occurs. Deadlocks in OS kernels are dangerous, because they can infinitely block the involved threads, causing performance degradation and even system hangs.

To find deadlocks, many existing approaches [5, 9–11, 15, 21, 27, 28, 31, 33, 44, 45, 55] dynamically monitor thread execution and lock-related operations to detect locking cycles. These approaches have shown promising results in both user-level applications and OS kernels. For example, Lockdep [33] is a widely-used lock-usage validator integrated in the Linux

kernel. It detects deadlocks, double locks and other locking issues, by dynamically tracking the state of each lock class and checking the dependencies between different lock classes. However, dynamic analysis approaches require well-constructed workloads or substantial test cases to cover the code containing bugs, and thus their detection coverage is often limited in runtime testing.

To improve detection coverage, some approaches [30, 41, 42, 46, 52] use static analysis to detect deadlocks in user-level applications. However, these approaches are ineffective in detecting deadlocks in OS kernels, for two main reasons. First, these approaches requires a fixed entry point (such as a `main` function) to start dataflow analysis; but an OS kernel consists of many kernel modules, each of which has no such a fixed entry point [4, 43]. Second, these approaches need to identify concurrent code and perform concurrency alias analysis according to thread-creation function calls (such as the calls to `pthread_create`) and related arguments; but the concurrency of OS kernel is often determined by the concurrent execution of specific interface functions in each kernel module [2], not explicitly calling thread-creation functions.

To our knowledge, RacerX [19] is the sole existing static analysis approach to systematically detect deadlocks in OS kernels. It uses *locking constraint* to describe the locking situation when each lock is acquired, e.g., if a code path acquires the locks A and then B , its locking constraint is $A \rightarrow B$. RacerX performs flow-sensitive and inter-procedural analysis to identify the code paths containing locking constraints (such code paths are referred to as *target code paths* subsequently) from OS kernel code, and then recursively compares between each two target code paths with their locking constraints to detect locking cycles as deadlocks.

However, RacerX still has some limitations. First, though using some heuristic techniques (like result ranking), RacerX still has a high false positive rate of 46%, due to neglecting the feasibility and concurrency of code paths. Second, RacerX neglects alias relationships, causing both false positives and negatives. Finally, RacerX simply compares between each two target code paths with their locking constraints in a recursive

way to detect locking cycles. This method works well for old OS kernels (like Linux 2.5.62 checked in its paper), but can be inefficient for modern OS kernels (like Linux 5.10 checked in our evaluation) that are much larger and more complex. Since the RacerX paper published in 2003, no new static approach has been proposed to systematically detect deadlocks in OS kernels. Thus, it is important to design a new static approach to perform effective deadlock detection in modern OS kernels.

In this paper, we design a practical static analysis approach named DLOS, to effectively detect deadlocks in OS kernels. DLOS consists of three key techniques:

(1) DLOS uses a *summary-based lock-usage analysis* to efficiently extract the code paths containing distinct locking constraints from kernel code. Our analysis uses function summaries to avoid repeated code analysis in the same functions, and it also drops the target code paths containing repeated locking constraints. To improve accuracy, our analysis is flow-sensitive and inter-procedural with the consideration of alias relationships, and it also uses a light-weight method to validate code-path feasibility with an SMT solver.

(2) DLOS uses a *reachability-based comparison method* to efficiently detect locking cycles from locking constraints. We observe that there are substantial target code paths containing distinct locking constraints in modern OS kernels (like Linux 5.10). Thus, when detecting locking cycles, simply comparing between each two target code paths with their locking constraints in a recursive way is quite time-consuming. To solve this problem, for each target code path, our method maintains a constraint reachability graph to store the locking constraints that are reachable starting the comparison from this code path, and the involved target code paths. By using constraint reachability graphs, our method can reduce repeated comparison of locking constraints, to improve the detection efficiency. If a locking cycle is found, it is considered as a possible deadlock, with the involved target code paths and locking constraints.

(3) DLOS uses a *two-dimensional filtering strategy* to effectively drop false positives, by validating the feasibility and concurrency of target code paths for each possible deadlock. On the one hand, as using an SMT solver to completely validate the feasibility of each code path is quite time-consuming, our strategy validates code paths in a phased way. Specifically, during locking-constraint extraction, as substantial code paths are required to be validated, making the validation efficiency more important, our strategy uses a simple and light-weight path-condition checking method to drop obviously infeasible code paths containing locking constraints; and then after locking-cycle detection, as the code paths of possible deadlocks should occupy a very small proportion of all the code paths, making the validation accuracy more important, our strategy uses a complete and heavy-weight path-condition checking method to drop false deadlocks. On the other hand, for each possible deadlock, our strategy checks the concurrency of its code paths, by analyzing their call graphs and looking for common locks acquired in these code paths.

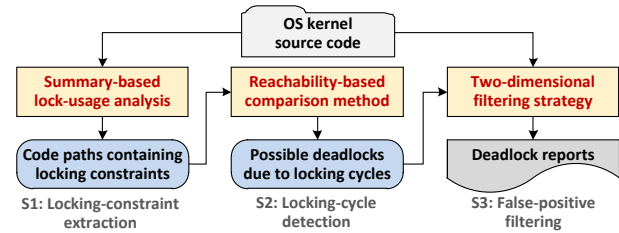


Figure 1: DLOS workflow.

Overall, DLOS has three main stages shown in Figure 1. In Stage 1, DLOS uses our summary-based lock-usage analysis to extract the code paths containing distinct locking constraints. In Stage 2, according to the extracted code paths and locking constraints, DLOS uses our reachability-based comparison method to detect possible deadlocks. In Stage 3, DLOS uses our two-dimensional filtering strategy to check possible deadlocks and drop false positives. After these stages, DLOS reports the final reports of the found deadlocks.

Compared to RacerX, DLOS has two main advantages. First, DLOS can achieve better accuracy than RacerX, by validating code-path feasibility with an SMT solver, considering alias relationships and checking the concurrency of the involved code paths for reported deadlocks. Second, DLOS can spend less time than RacerX, by extracting and comparing locking constraints more efficiently.

We have implemented DLOS with LLVM [32] and Z3 [54]. DLOS performs automated static analysis on the LLVM bytecode of the checked OS kernel. Overall, we make three main contributions in this paper:

- We analyze the challenges of static deadlock detection in OS kernels, and propose three key challenges to address these challenges: (1) a *summary-based lock-usage analysis* to efficiently extract the code paths containing distinct locking constraints from kernel code; (2) a *reachability-based comparison method* to efficiently detect locking cycles from locking constraints; (3) a *two-dimensional filtering strategy* to effectively drop false positives by validating code-path feasibility and concurrency.
- Based on these three key techniques, we design a practical static analysis approach named DLOS, to effectively detect deadlocks in OS kernels.
- We evaluate DLOS on Linux 4.9 and 5.10, and find 46 and 65 deadlocks, respectively. We manually check these deadlocks, and find that 39 and 54 deadlocks are real. 21 of the real deadlocks found in Linux 4.9 have been fixed in Linux 5.10. We have reported the 54 real deadlocks found in Linux 5.10 to Linux kernel developers, and 31 of them have been confirmed.

The rest of this paper is organized as follows. Section 2 introduces the background and motivation. Section 3 introduces the challenges of static deadlock detection in OS kernels and our key techniques to address these challenges. Section 4 introduces DLOS. Section 5 shows our evaluation. Section 6

makes a discussion about DLOS. Section 7 presents related work, and Section 8 concludes this paper.

2 Background and Motivation

We first introduce deadlock and its detection, then explain the concurrency model of OS kernels, and finally motivate our work using a real deadlock in the Linux kernel.

2.1 Deadlock and Its Detection

To protect critical data from concurrent accesses, several kinds of synchronization primitives are designed and used. Locks are the most frequently-used synchronization primitives in real-world programs, to guarantee atomicity and prevent data races. However, if locks are incorrectly used, a deadlock can occur when one thread holds a lock that other concurrent threads want to acquire and vice versa.

Locking cycles in concurrent threads can cause deadlocks. The most common case is the ABBA lock in two threads, as shown in Figure 2(a). Namely, one thread acquires the locks A and then B ($A \rightarrow B$), while the other concurrent thread acquires the locks B and then A ($B \rightarrow A$), causing a locking cycle ($A \rightarrow B, B \rightarrow A$). In three or more threads, deadlocks can also occur due to such locking cycles, as shown in Figure 2(b).

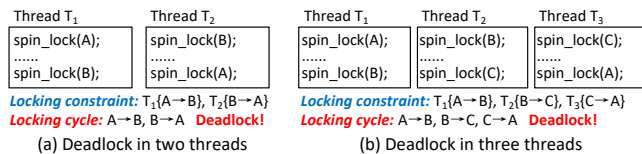


Figure 2: Deadlock examples.

For dynamic analysis, deadlock detection has two basic steps, namely extracting locking constraints in concurrent threads and then comparing these locking constraints to detect locking cycles. For static analysis, deadlock detection is similar but more complex. On the one hand, without exact runtime information about thread execution, static analysis has to identify locking constraints from each code path and validate the concurrency of code paths. On the other hand, without exact values of accessed variables, static analysis has to validate the feasibility of code paths using an SMT solver.

2.2 Concurrency Model of the OS Kernel

A modern OS kernel consists of many kernel modules, including filesystems, network modules, device drivers, etc. Each kernel module has some specific interface functions that are called by upper-level programs, including other kernel modules via function-pointer calls and user-level applications via system calls. Figure 3 shows some examples of interface functions that are assigned to function-pointer fields. These interface functions form the entry points of the kernel module,

```
FILE: linux-5.10/fs/gfs/file.c
1108. file_operations gfs2_file_fops = {
1109.     .llseek = gfs2_llseek,
1110.     .read_iter = generic_file_read_iter,
1111.     .write_iter = gfs2_file_write_iter,
1112.     .unlocked_ioctl = gfs2_ioctl,
1123. }

FILE: linux-5.10/drivers/net/wan/lmc_main.c
808. struct net_device_ops lmc_ops = {
809.     .ndo_open = lmc_open,
810.     .ndo_stop = lmc_close,
811.     .ndo_change_mtu = hdlc_change_mtu,
812.     .ndo_start_xmit = hdlc_start_xmit,
816. }
```

Figure 3: Examples of interface functions in kernel modules.

and all other functions defined in the kernel module are called by them [4, 43]. Due to this execution model, the concurrency of OS kernel is often determined by the concurrent execution of specific interface functions in each kernel module [2]. In fact, a kernel module can also explicitly call thread-creation functions (such as `kthread_create` in the Linux kernel), but such operations are not common in kernel module code.

Different from the OS kernel, each user-level application has a fixed entry point (like a `main` function) and explicitly calls thread-creation functions (like `pthread_create`) to start concurrent execution. Accordingly, to detect deadlocks in user-level applications, existing static approaches [30, 41, 42, 46, 52] start dataflow analysis from this fixed entry point, and identify concurrent code for concurrency alias analysis according to thread-creation function calls and related arguments. Due to the difference between the concurrency models of OS kernels and user-level applications, these approaches are ineffective in detecting deadlocks in OS kernels.

2.3 Motivating Example

Figure 4 presents a real and already fixed deadlock in the `btrfs` filesystem, and this bug is found by our approach DLOS in the evaluation of checking Linux 4.9. When the function `btrfs_read_chunk_tree` is executed on the code path $P1$, it acquires the locks `root->fs_info->chunk_mutex` and then `orig->device_list_mutex`; when the function `btrfs_remove_chunk` is executed on the code path $P2$, it acquires the locks `fs_devices->device_list_mutex` and then `root->fs_info->chunk_mutex`. During filesystem execution, the functions `btrfs_read_chunk_tree` and `btrfs_remove_chunk` are able to be concurrently executed at runtime, and the locks `orig->device_list_mutex` and `fs_devices->device_list_mutex` can be identical, and thus an ABBA deadlock can occur. This deadlock was introduced by the commit `57ba4cb85bff` [16] in Linux 4.7, and it was found by Lockdep [33] and fixed by the commit `01d01caf19ff` [17] in Linux 5.9, after over 4 years later. In fact, Lockdep is integrated in the Linux kernel for dynamic deadlock detection, but it took Lockdep such a long time to find this deadlock, because the interleaving of the code paths $P1$ and $P2$ are infrequent in real execution.

This example illustrates why deadlocks occur in OS kernels. First, determining concurrent code paths and identifying the same locks in these code paths require substantial knowledge of OS kernels. In the example, without deep understanding of filesystems and extensive testing, it may

```

Code Path P1:
// FILE: linux-4.9/fs/btrfs/volumes.c
btrfs_read_chunk_tree
-> lock_chunks [Line 6803]
-> mutex_lock(&root->fs_info->chunk_mutex) [Line 517]
-> read_one_dev [Line 6833]
-> open_seed_devices [Line 6601]
-> clone_fs_devices [Line 6558]
-> mutex_lock(&orig->device_list_mutex) [Line 734]
      A→B
Code Path P2:
// FILE: linux-4.9/fs/btrfs/volumes.c
btrfs_remove_chunk
-> mutex_lock(&fs_devices->device_list_mutex) [Line 2844]
-> lock_chunks [Line 2857]
-> mutex_lock(&root->fs_info->chunk_mutex) [Line 517]
      B→A

```

Figure 4: A real deadlock in Linux 4.9 *btrfs* filesystem.

be difficult to know code paths *P1* and *P2* can be concurrently executed, and the locks `orig->device_list_mutex` and `fs_devices->device_list_mutex` can be identical. Second, incorrect fixing of known bugs can introduce new and hard-to-find concurrency bugs. In the example, the commit 57ba4cb85bff introducing the deadlock aimed to fix a harmful data race in the functions `btrfs_remove_chunk` and `btrfs_dev_replace_finishing`, but this commit incautiously introduces a locking cycle in the functions `btrfs_remove_chunk` and `btrfs_read_chunk_tree`. Finally, multiple functions (including concurrent functions and the functions called by them) and variables in these functions need to be considered.

By scanning the reported deadlocks in the Linux kernel, we find that most of them are found in stress testing and kernel fuzzing. But the detection coverage of runtime testing heavily relies on the provided workloads, causing many real deadlocks to be missed. Static analysis can conveniently achieve high detection coverage without actual execution of the OS kernel. However, as the sole existing static approach of systematically detecting deadlocks in OS kernels, RacerX [19] still has many false positives, and its concurrency analysis can be inefficient to modern OS kernels (like Linux 5.10 checked in our evaluation) that are much larger and more complex than old OS kernels (like Linux 2.5.62 checked in the RacerX paper). Thus, it is important to design a new static approach to perform effective deadlock detection in modern OS kernels.

3 Challenges and Key Techniques

To detect deadlocks, static analysis needs to first extract the code paths containing distinct locking constraints (such code paths are referred to as *target code paths* subsequently) from kernel code, and then compare these code paths with their locking constraints to detect locking cycles as deadlocks. However, performing these steps for checking OS kernel code has three main challenges:

C1: Extracting locking constraints. A modern OS kernel is very large and complex, because it has many kernel modules and lots of functions with complicated call graphs. Thus, extracting locking constraints in OS kernel code can be quite time-consuming and inaccurate.

C2: Detecting locking cycles. Due to the large and complex code base of the OS kernel, there are substantial target code paths containing distinct locking constraints. Thus, when detecting locking cycles, simply comparing between each two target code paths with their locking constraints in a recursive way is quite inefficient.

C3: Dropping false bugs. On the one hand, without validating the feasibility of code paths, static analysis can extract many infeasible target code paths and thus report many false bugs. On the other hand, each deadlock involves two or more target code paths that should be able to concurrently executed. Thus, without validating the concurrency of these target code paths, static analysis can report many false bugs whose target code paths cannot be concurrently executed.

To solve the above challenges, we propose three key techniques. For *C1*, we propose a *summary-based lock-usage analysis* to efficiently extract the code paths containing distinct locking constraints from kernel code. For *C2*, we propose a *reachability-based comparison method* to efficiently detect locking cycles from locking constraints. For *C3*, we propose a *two-dimensional filtering strategy* to effectively drop false positives by validating code-path feasibility and concurrency. We will introduce these techniques as follows.

3.1 Summary-Based Lock-Usage Analysis

Our summary-based lock-usage analysis has two basic stages: (S1) performing a dataflow analysis to collect target code paths containing distinct lock-acquire/release operations; and then (S2) performing a static lockset analysis to compute locking constraints for each target code path.

S1: Collecting target code paths. In this stage, the dataflow analysis has some properties: 1) this analysis is flow-sensitive and inter-procedural with the consideration of alias relationships, which can improve the accuracy; 2) this analysis uses function summaries to reduce repeated analysis, which can improve the efficiency; 3) this analysis drops the target code paths containing repeated lock-acquire/release operations, which can reduce repeated comparison in locking-cycle detection; 4) this analysis uses a light-weight method to validate the feasibility of each analyzed code path, which can reduce false positives of deadlock detection. This dataflow analysis traverses the code paths in the analyzed function *func*.

Figure 5 shows the main procedure of this dataflow analysis, which is represented as *DataFlowAnalysis*. It creates a function summary *func_sum*, which stores basic information about *func* (like function name and function-definition location) and the target code paths in *func*. This function summary is initialized with no target code path (line 1). This analysis handles each code path *code_path* in *func* with three steps (lines 2-30). First, it creates a data structure *tar_path* to collect the lock-acquire/release function calls and analyzed basic blocks in *code_path* (line 3), and then checks each function call *call* in *code_path* (lines 4-26). If *call* is used to acquire


```

DataFlowAnalysis(func)
Input: func – the analyzed function
Output: func_sum – function summary storing basic information about func
and the target code paths in func
-----
1: func_sum->tar_path_set :=  $\emptyset$ ;
2: foreach code_path in GetCodePathSet(func) do
3:   tar_path := CreateTargetCodePath(code_path);
4:   foreach call in GetCallSetInPath(code_path) do
5:     called_func := GetCalledFunction(call);
6:     if CheckLockFunction(called_func) then
7:       AddLockVector(call, tar_path->lock_vec);
8:     else
9:       // Use function summary to reduce repeated analysis
10:      called_func_sum := FindFuncSummary(called_func);
11:      if called_func_sum == NULL then
12:        called_func_sum := DataFlowAnalysis(called_func);
13:      end if
14:      // Top-down analysis of all target code paths in the callee
15:      called_tar_path_set := called_func_sum->tar_path_set;
16:      foreach called_tar_path in called_tar_path_set do
17:        tar_path_tmp := SplicePathInfo(tar_path, called_tar_path);
18:        if LightPathCheck(tar_path_tmp) == TRUE then
19:          AddPathSet(tar_path_tmp, func_sum->tar_path_set);
20:        end if
21:      end foreach
22:      // Bottom-up analysis of one target code path selected in the callee
23:      rand_tar_path := RandomSelect(called_tar_path_set);
24:      tar_path := SplicePathInfo(tar_path, rand_tar_path);
25:    end if
26:  end foreach
27:  if LightPathCheck(tar_path) == TRUE then
28:    AddPathSet(tar_path, func_sum->tar_path_set);
29:  end if
30: end foreach
31: DropRepeatTargetCodePath(func_sum->tar_path_set);
32: return func_sum;
-----

```

Figure 5: Dataflow analysis of collecting target code paths.

or release a lock, it is added into the lock-operation vector *lock_vec* of *tar_path* (line 7); otherwise this analysis handles its called function *called_func*. If *called_func* has been already analyzed, its function summary is gotten and stored as *called_func_sum* (line 10); otherwise, *DataFlowAnalysis* is recursively used to compute its function summary as *called_func_sum* (line 12). Then, from *called_func_sum*, this analysis gets and handles the target code paths in *called_func* (lines 15-21) in a top-down manner. For each such target code path *called_tar_path*, this analysis splices it with *tar_path* to form a new and possible target code path *tar_path_tmp*. This analysis uses a light-weight method (will be explained in Section 3.3) to validate the code-path feasibility of *tar_path_tmp*; if the code path is feasible, *tar_path_tmp* is considered as a possibly real target code path in *func* and added into the function summary *func_sum*->*tar_path_set* (lines 18-20). To avoid the explosion of bottom-up code paths from the callee function *called_func*, this analysis randomly selects one of the target code paths in this function and splices it into *tar_path* (lines 23-24). Before the code path ends, this analysis uses the light-weight method again to validate the code-path feasibility of *tar_path*; if the code path is feasible, *tar_path* is considered as a possibly real target code path in *func* and added into the function summary *func_sum*->*tar_path_set* (lines 27-29). After handling each code path, this analysis checks *func_sum*->*tar_path_set* to drop the target code paths containing identical lock-operation vectors (line 31), which can

reduce repeated comparison of target code paths in locking-cycle detection. Finally, this analysis returns the function summary *func_sum* (line 32), which can be used to analyze other functions that call *func*.

Besides the procedure shown in Figure 5, this dataflow analysis also performs an intra-procedural, flow-insensitive and Andersen-style alias analysis [1] to identify all variables aliased with the lock argument of each lock-acquire/release function call. The alias analysis can help to improve the accuracy of computing locking constraints in S2. Moreover, each function summary stores the information about arguments and global variables, and drops the information about local variables, which are never used outside the function.

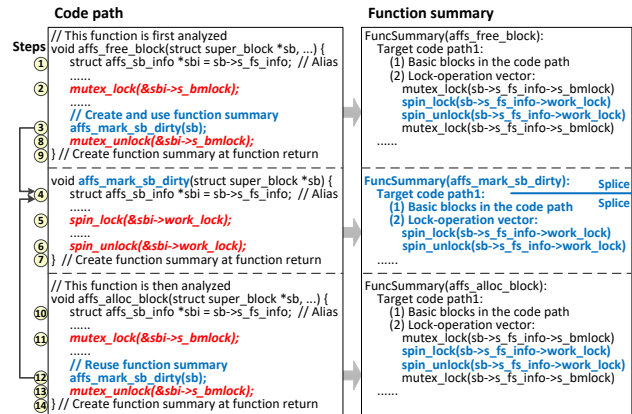


Figure 6: Example of summary-based dataflow analysis.

Example. We illustrate this dataflow analysis using the simplified code of the Linux *affs* filesystem in Figure 6. This figure shows three functions and partial code paths of them. According to the function position order, this analysis first analyzes the function *affs_free_block* and then *affs_alloc_block*, the analysis steps are represented as ①. This dataflow analysis also considers the alias relationships at ①, ④ and ⑩. At ③, there is a function call to *affs_mark_sb_dirty*, so this dataflow analysis first handles this function, then creates its function summary, and finally splices the target code path of this function summary into the analyzed target code path of *affs_free_block*. At ⑫, the function *affs_mark_sb_dirty* is called again, so its function summary is reused, and the target code path of this function summary is spliced into the analyzed target code path of *affs_alloc_block*. By using the function summary, the analysis efficiency can be effectively improved.

Note that to avoid path explosion caused by bottom-up code paths of each callee function, this dataflow analysis uses a partial bottom-up analysis. Specifically, it randomly selects one of the target code paths in its function summary, and splices this path into the analyzed target code in the caller function, as shown at lines 23-24 in Figure 5. However, this method can miss other target code paths of the callee function, which can cause false negatives of deadlock detection. Even

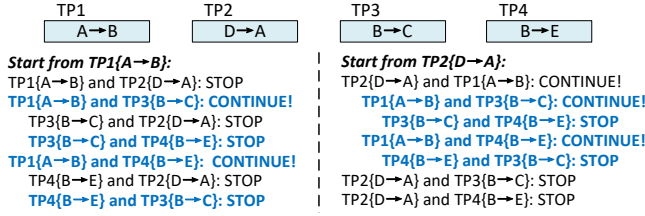


Figure 7: Example of the traditional comparison.

so, compared to RacerX [19] that only has top-down analysis without bottom-up analysis, this dataflow analysis is more accurate by using partial bottom-up analysis. In the future, we will implement a more complete and low-complexity bottom-up analysis, by referring to some existing approaches [39, 40].

S2: Computing locking constraints. This step uses a static lockset analysis to compute locking constraints in the target code paths collected in S1. This lockset analysis is similar to dynamic lockset analysis proposed in Eraser [47] for race detection, but in a static way. For each target code path, this lockset analysis maintains a lockset storing the held locks, and it handles lock-acquire and -release function calls.

When encountering a lock-acquire function call, this analysis first creates and adds related locking constraints in the analyzed target code path, according to the locks in the lockset and the acquired lock of this call; and then it adds this acquired lock into the lockset. For example, when this analysis handles the function call acquiring the lock X , if the lockset LS stores the held locks A and B , it first creates two locking constraints $A \rightarrow X$ and $B \rightarrow X$, then adds these locking constraints into the analyzed target code path, and finally adds X into LS . When encountering a lock-release function call, this analysis looks for and drops the involved lock in the lockset.

3.2 Reachability-Based Comparison Method

After extracting target code paths, we need to compare them to detect locking cycles as possible deadlocks. During comparison, we use a field-based analysis to identify the same locks in different code paths, if the lock variables' data structure types and fields are identical, which is similar to RacerX [19] and DCUAF [2]. Moreover, because a locking cycle can involve multiple target code paths (like the example deadlock in Figure 2(b)), the traditional method (used by existing static approaches like RacerX [19]) starts the comparison from each locking constraint in each target code path, and then recursively compares between each two target code paths with their locking constraints. Specifically, this method compares the current locking constraint with each locking constraint of each unhandled target code path. If they are matched, the current locking constraint is replaced with the matched locking constraint, and the comparison continues; if they are not matched, the comparison selects other target code paths. If all target code paths have been handled, the comparison stops. Once a locking cycle is found, this method reports a deadlock.

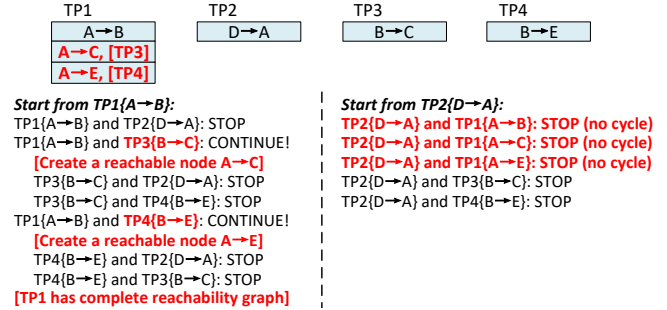


Figure 8: Example of our reachability-based comparison.

Example with the traditional comparison. We illustrate this traditional method using an example in Figure 7, containing four target code paths ($TP1$, $TP2$, $TP3$ and $TP4$), each of which contains one locking constraint. This method first starts the comparison from the locking constraint $A \rightarrow B$ in $TP1$ (namely $TP1\{A \rightarrow B\}$), and then starts the comparison from the locking constraint $D \rightarrow A$ in $TP2$ (namely $TP2\{D \rightarrow A\}$). The detailed comparison steps are also shown in the figure.

In Figure 7, we find that when starting the comparison from $TP2\{D \rightarrow A\}$, some steps (marked in blue and bold font) perform repeated comparison that has been done when starting from $TP1\{A \rightarrow B\}$. In fact, such repeated comparison of locking constraints are common in locking cycle detection, because many code paths handle the same locks but have different locking orders, leading to different locking constraints. Thus, if such repeated comparison can be reduced, the locking cycle detection can be much more efficient.

Based on this idea, we propose a reachability-based method to efficiently compare locking constraints for locking-cycle detection. During comparison, this method maintains a constraint reachability graph for the target code path that the comparison starts from. This reachability graph contains some reachable nodes, each of which indicates an *indirect* locking constraint used for subsequent comparison:

$$\bigwedge_{i=1}^n (TP_i\{A_i \rightarrow A_{i+1}\}) \Rightarrow TP_{indirect}\{A_1 \rightarrow A_{n+1}, TP_{set}\}$$

$$TP_{set} = \{TP_1, TP_2, \dots, TP_n\}$$

This indirect locking constraint is added in the handled target code path. When our method finishes the comparison starting from all the locking constraints in this target code path, its reachability graph is completely built. The indirect locking constraints in this reachability graph are used to reduce repeated comparison involving the handled target code path. Specifically, if any locking constraint (direct or indirect) in this target code path is matched, the comparison stops and checks whether there is a locking cycle. Note that our method assumes a target code path is never concurrently executed with itself, because static analysis has insufficient information to infer whether a code path can be concurrently executed with itself. This assumption is also followed by existing static approaches, such as RacerX [19] and DCUAF [2].

Example with our reachability-based comparison. To illustrate our method, we still use the example in Figure 7. The key steps performed by our method are marked in red and bold font. Our method still first starts the comparison from the locking constraint $TP1\{A \rightarrow B\}$. Because $TP1\{A \rightarrow B\}$ matches $TP3\{B \rightarrow C\}$ and $TP4\{B \rightarrow E\}$, our method creates two indirect locking constraints $TP1\{A \rightarrow C, [TP3]\}$ and $TP1\{A \rightarrow E, [TP4]\}$ and adds them in the target code path $TP1$. After $TP1$ is handled, its reachability graph is completely built. When our method starts the comparison from the locking constraint $TP2\{D \rightarrow A\}$, the three locking constraints (including two indirect ones) in $TP1$ are matched. Because the reachability graph of $TP1$ is complete, the comparison stops when these locking constraints are handled, which can avoid the repeated steps performed by the traditional comparison method in Figure 7. In this way, our method can effectively reduce the time usage of locking-cycle detection.

Besides, for each indirect locking constraint, our method also stores the related original locking constraints and target code paths. During comparison, if a locking cycle is found as a possible deadlock, our method can conveniently recover the information about the involved locks and their code paths, which is used for false-positive filtering in Section 3.3.

3.3 Two-Dimensional Filtering Strategy

For a possible deadlock, our strategy checks whether it is a false positive, in two dimensions, namely validating the feasibility and concurrency of its target code paths.

D1: Validating code-path feasibility. For a given code path, we can use an SMT solver to validate the satisfiability of all the branch conditions and variable accesses (including read and write operations) in this code path. For deadlock detection, there are two possible stages where code-path validation can be performed: (S1) during the dataflow analysis extracts target code paths in Section 3.1, we can validate the feasibility of each extracted target code path; (S2) after locking-cycle detection in Section 3.2, we can validate the feasibility of the target code paths for each possible deadlock.

In S1, because the dataflow analysis needs to handle substantial code paths, if we perform complete and accurate validation of these code paths, the time cost will be quite high. In S2, we believe that the code paths of possible deadlocks should occupy a very small proportion of all the target code paths extracted in the dataflow analysis, and thus it is acceptable to perform complete and accurate validation of these code paths. An alternative way is to just perform code-path validation in S2. However, without the validation in S1, lots of infeasible target code paths will be extracted for locking-cycle detection, which can also introduce high time cost.

Based on the above consideration, our strategy uses a staged and balanced way. In S1, our strategy uses a simple and light-weight path-condition checking method to efficiently check the extracted target code paths. This method checks only

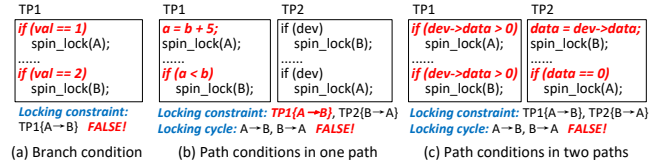


Figure 9: Examples of code-path feasibility validation.

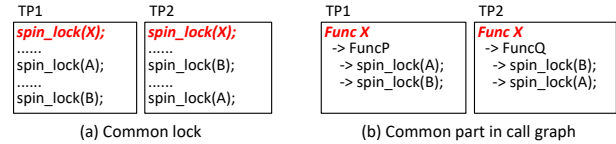


Figure 10: Examples of code-path concurrency checking.

branch conditions in each code path, without handling variable accesses outside branch conditions. Thus, this method is fast but relatively inaccurate, and it can quickly drop many target code paths that are obviously infeasible. Figure 9(a) shows an example target code path that can be dropped by this method. In S2, our strategy uses a complete and heavy-weight path-condition checking method to accurately check the code paths of possible deadlocks. This method checks both branch conditions and variable accesses in each code path. Thus, this method is relatively slow but accurate, and it can effectively drop false deadlocks involving complex path conditions.

In fact, because a deadlock contains two or more code paths that are interleaved in concurrent execution, these code paths may access some shared variables that should have identical values. Thus, for each possible deadlock, the heavy-weight method in S2 performs code-path validation in two ways. First, for each code path of this deadlock, the method validates its feasibility; if any code path is identified to be infeasible by an SMT solver, this deadlock is considered to be false and dropped. Second, the method extracts shared variables having identical data structure types and fields in each two code paths, then identifies the variable accesses and branch conditions that are related to these shared variables in the code paths, and finally translates the identified operations into SMT constraints of an SMT solver. If these SMT constraints are computed to be unsatisfiable, this deadlock is considered to be false and dropped. Figure 9(b) and Figure 9(c) show two example false deadlocks that can be dropped in these two ways, respectively.

D2: Checking code-path concurrency. For a deadlock, its target code paths should be able to be concurrently executed; otherwise, this deadlock is false. For each possible deadlock, our strategy checks the concurrency of its target code paths in two ways. First, our strategy checks whether there is a common lock acquired before the involved lock-acquire operations in any two of the target code paths. If so, these code paths cannot be concurrently executed, so this possible deadlock is considered to be false and dropped. Second, our strategy extracts the call graph of each target code path, and checks whether any two of these call graphs have common parts. If so,

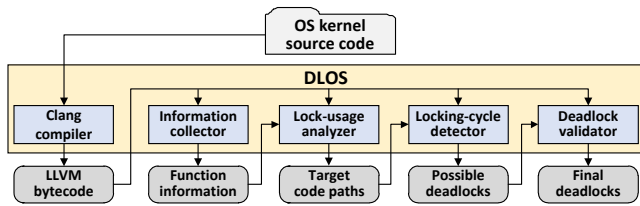


Figure 11: DLOS architecture.

it indicates that the related two code paths may be sequentially executed at different time points of the same thread, so this possible deadlock is considered to be false and dropped. Figure 10(a) and Figure 10(b) show two example false deadlocks that can be dropped in these two ways, respectively.

4 DLOS Approach

Based on the three key techniques in Section 3, we design a practical static approach named DLOS, to detect deadlocks in OS kernels. We have implemented DLOS with Clang [13] and Z3 [54]. DLOS automatically performs static analysis on the LLVM bytecode files of the OS kernel. Figure 11 shows the architecture of DLOS, which has four phases:

P1: Source-code compilation. The *Clang compiler* compiles the kernel source files into LLVM bytecode files, and then the *information collector* handles each function in LLVM bytecode to collect the function’s information (including function name, function-definition position, etc.). The information is used for inter-procedural analysis across source files.

P2: Locking-constraint extraction. The *lock-usage analyzer* uses our summary-based lock-usage analysis to handle LLVM bytecode files. This analysis starts at the entry of each function in the kernel code, to extract target code paths containing distinct locking constraints.

P3: Locking-cycle detection. The *locking-cycle detector* uses our reachability-based comparison method to check the extracted target code paths with locking constraints, and detects locking cycles as possible deadlocks. We observe that a kernel module often acquires private locks that are not accessible for other kernel modules, and thus the detector focuses on checking target code paths in the same kernel module.

P4: False-positive filtering. The *deadlock validator* uses our two-dimensional filtering strategy to check possible deadlocks and drop false positives. Besides, two possible deadlocks may have identical problematic locking operations but differ in code paths. To drop such repeated bugs, for a new possible deadlock, the validator checks whether it has the same problematic locking operations with any already detected deadlock; if so, it is considered to be repeated and dropped.

Implementation details. DLOS performs lock-usage analysis and path validation from the entry of each function in OS code, so it can handle different execution contexts like interrupt handling. However, DLOS does not handle function-pointer calls

at present, so it cannot detect deadlocks across kernel modules connected by function pointers. Besides, DLOS cannot analyze RCU locks, as RCU lock-acquiring/release functions (like `rcu_read_lock` and `rcu_read_unlock`) have no argument. Finally, to accelerate deadlock detection, DLOS can support the parallelism of handling multiple kernel modules using multi-thread execution.

5 Evaluation

To validate the effectiveness of DLOS, we evaluate it on the Linux kernel. To cover different kernel versions, we select an old version 4.9 and a recent version 5.10. Table 1 shows the basic information about these kernel versions, and source code lines are counted by CLOC [14]. We run the experiments on a regular x86-64 PC with eight Intel i7-3770@3.40G CPUs and 16GB memory. We use the kernel configuration *allyesconfig* to enable all kernel code for the x86-64 architecture.

Description	Linux 4.9	Linux 5.10
Release time	December 2016	December 2020
Source files (.c)	23.7K	29.4K
Source code lines (.c)	11.4M	14.7M

Table 1: Basic information about the checked OS kernels.

5.1 Bug Detection

We configure DLOS with common lock-acquiring/release functions (like `spin_lock` and `spin_unlock`) according to the Linux kernel documents [34], and then run DLOS to automatically check the kernel source code. We manually check all the deadlocks found by DLOS to identify real bugs. Table 2 shows the results, and we have the following findings:

Code analysis. DLOS can scale to large code bases of OS kernels. Specifically, it analyzes 8.5M and 11.7M source code lines in 14.5K and 19.9K source files in Linux 4.9 and 5.10, respectively, within 7 hours. The remaining 2.9M and 3.0M source code lines in 9.2K and 9.5K source files are not analyzed, because they are not enabled by *allyesconfig* for the x86-64 architecture.

Efficiency improvement. DLOS improves the analysis efficiency from two aspects:

First, when extracting target code paths containing locking constraints, our lock-usage analysis uses function summaries to reduce repeated code analysis in the same functions. Specifically, around 93% of the times DLOS handles a function call is able to reuse an existing function summary, without the need of analyzing the function’s definition again.

Second, when detecting locking cycles, our reachability-based comparison method uses constraint reachability graphs to reduce repeated comparison of locking constraints. Specifically, with 196K and 222K indirect locking constraints created by our method, 851K and 946K times of repeated comparison are reduced in Linux 4.9 and 5.10, respectively.

	Description	Linux 4.9	Linux 5.10
<i>Lock-usage analysis</i>	Source files (analyzed/all)	14.5K/23.7K	19.9K / 29.4K
	Source code lines (analyzed/all)	8.5M/11.4M	11.7M/14.7M
	Times of handling functions	4,102K	5,032K
	Times of reusing function summaries	3,816K	4,682K
	Extracted distinct target code paths	102K	117K
	Extracted locking constraints	323K	439K
<i>Locking-cycle detection</i>	Created indirect locking constraints	196K	222K
	Times of reducing comparison	851K	946K
	Possible deadlocks	465	539
<i>False-positive filtering</i>	Dropped infeasible target code paths	464K	524K
	False bugs due to infeasible paths	220	258
	False bugs due to common locks	78	94
	False bugs due to call graphs	101	122
	Total false bugs	419	474
<i>Deadlock</i>	Found bugs	46	65
	Real bugs	39	54
<i>Time usage</i>	Lock-usage analysis	265m	294m
	Locking-cycle detection	85m	96m
	False-positive filtering	22m	28m
	Total time	372m	418m

Table 2: Deadlock-detection results.

False-positive dropping. DLOS uses our two-dimensional filtering strategy to drop false positives from three aspects:

First, when extracting target code paths containing locking constraints, our strategy uses a simple and light-weight code-path validation method to drop 464K and 524K infeasible target code paths in Linux 4.9 and 5.10, respectively. In addition, by dropping these target code paths, the related unnecessary locking-constraint comparison can be avoided in locking-cycle detection, which also improves the efficiency of deadlock detection.

Second, after locking-cycle detection reports possible deadlocks, our strategy uses a complete and heavy-weight code-path validation method to drop 220 and 258 false bugs in Linux 4.9 and 5.10, respectively. Indeed, these false bugs' code paths are failed to be dropped in our lock-usage analysis, because the light-weight code-path validation method used in this analysis is efficient but relatively inaccurate. To improve accuracy, the heavy-weight code-path validation method completely checks both branch conditions and variable accesses in the code paths of each possible deadlock, and thus it successfully drops these false bugs after locking cycle detection.

Finally, our strategy checks the concurrency of possible deadlocks, and drops 179 and 216 false bugs in Linux 4.9 and 5.10, respectively, because their target code paths are considered to be non-concurrent. Specifically, 78 and 94 bugs are dropped due to holding a common lock in target code paths; 101 and 122 bugs are dropped due to containing common parts in the call graphs of target code paths.

Deadlock finding. DLOS reports 46 and 65 deadlocks in Linux 4.9 and 5.10, respectively. We spent eight hours on checking all these 111 reported deadlocks. We identify 39 and 54 deadlocks are real in Linux 4.9 and 5.10, respectively. 21 real deadlocks in Linux 4.9 have been fixed in Linux 5.10, including the deadlock in the *btfs* filesystem shown in Figure 4. Thus, DLOS can find known deadlocks. Moreover, we have reported the 54 real deadlocks in Linux 5.10 to Linux

kernel developers, and 31 of them have been confirmed. We are still waiting for the reply of the remaining ones. Thus, DLOS can find new deadlocks.

We infer that these real deadlocks are missed by Lockdep in extensive kernel testing, because their thread interleavings are infrequent to occur, and constructing workloads to cover these thread interleavings is difficult. Thus, DLOS can indeed find many deadlocks that are hard to find in runtime testing.

Besides, we believe that DLOS is helpful to deadlock reproduction, because it produces the detailed code paths of the found deadlocks. With these code paths, time delays can be strategically injected and carefully controlled, to cover specific thread interleavings and reproduce the found deadlocks. We have manually performed this way for several deadlocks in kernel modules that we can run, including the two deadlocks in Figure 4 and Figure 12(a), and these deadlocks can be successfully reproduced at runtime.

Deadlock details. Among the 93 real deadlocks in Linux 4.9 and 5.10, 78 (33 in Linux 4.9 and 45 in Linux 5.10) occur in device drivers, and 15 (6 in Linux 4.9 and 9 in Linux 5.10) occur in filesystems. This result indicates that device drivers remain a significant source of OS bugs [49]. Besides, for 86 deadlocks (35 in Linux 4.9 and 51 in Linux 5.10), DLOS reports two code paths for each of them, indicating it is caused by two locks in two threads; for the remaining 7 deadlocks, DLOS reports three code paths for each of them, indicating it is caused by three locks in three threads.

5.2 False Positives and Negatives

False positives. DLOS reports 7 and 11 false bugs in Linux 4.9 and 5.10, resulting the false positives rates of 15% and 17%, respectively. By manually checking these false bugs, we find that they are reported for three main reasons:

First, the field-based analysis in locking-cycle detection can make mistakes when identifying the same locks in different code paths. This analysis identifies the same locks if the locks variables have the same data structure types and fields; but two different lock variables can also have the same data structure types and fields, and their data structure variables are different. This analysis cannot handle such cases at present. This reason causes DLOS to report 3 and 5 false bugs in Linux 4.9 and 5.10, respectively.

Second, although DLOS uses Z3 to validate path feasibility, it can still make mistakes when handling some complex cases, such as complicated arithmetic conditions and data dependence across multiple functions. This reason causes DLOS to report 2 and 3 false bugs in Linux 4.9 and 5.10, respectively.

Finally, the alias analysis in our lock-usage analysis is intra-procedural and flow-insensitive, and thus can identify wrong alias relationships across function calls, causing mistakes in locking-constraint extraction. This reason causes DLOS to report 2 and 3 false bugs in Linux 4.9 and 5.10, respectively.

False negatives. DLOS may still miss some real deadlocks for three main reasons:

First, our lock-usage analysis performs incomplete bottom-up analysis of each callee function, to avoid path explosion of inter-procedural analysis. Specifically, it randomly selects one of the target code paths in the callee function, and splices it into the analyzed target code in the caller function. Although the other target code paths in the callee function are handled in top-down analysis, they are neglected in bottom-up analysis, causing some locking constraints in the target code paths of the caller function to be missed.

Second, DLOS does not analyze function-pointer calls, and thus it cannot build complete call graphs for inter-procedural analysis. As a result, DLOS may miss real deadlocks involving the code that is reached through function-pointer calls.

Finally, DLOS considers that a target code path is never concurrently executed with itself. Indeed, to reduce false positives, DLOS validates two code paths' concurrency by checking their common locks and call graphs. However, this validation is infeasible for two identical code paths, and thus DLOS does not detect deadlocks occurring in the same target code path of different execution contexts.

5.3 Case Studies of the Found Deadlocks

Figure 12 shows two deadlocks found by DLOS in Linux 5.10, and they have been confirmed by Linux kernel developers.

Deadlock in SysRq command handling for filesystems. In Figure 12(a), when the function `do_thaw_all_callback` is executed on the code path *P1*, it first acquires the read-write semaphore `sb->s_umount` and then the mutex lock `bdev->bd_fsfreeze_mutex`; when the function `freeze_bdev` is executed on the code path *P2*, it first acquires the mutex lock `bdev->bd_fsfreeze_mutex` and then the read-write semaphore `sb->s_umount`. During SysRq commands [50] are handled for filesystems, the functions `do_thaw_all_callback` and `freeze_bdev` can be concurrently executed, and thus an ABBA deadlock can occur.

Deadlock in the LPFC SCSI driver. In Figure 12(b), when the function `lpfc_nvmet_unsol_fcp_issue_abort` is executed on the code path *P1*, it acquires the spinlocks `ctxp->ctxlock` and then `phba->sli4_hba.abts_nvmet_buf_list_lock`; when the function `lpfc_sli4_nvmet_xri_aborted` is executed on the code path *P2*, it acquires the spinlocks `phba->sli4_hba.abts_nvmet_buf_list_lock` and then `ctxp->ctxlock`. During driver execution, the functions `lpfc_nvmet_unsol_fcp_issue_abort` and `lpfc_sli4_nvmet_xri_aborted` can be concurrently executed, and thus an ABBA deadlock can occur.

From the feedback of kernel developers, the confirmed deadlocks found by DLOS require infrequent and special test cases to find at runtime, which indicates that DLOS is useful to detecting hard-to-trigger deadlocks via static analysis.

```
Code Path P1:
// FILE: linux-5.10/fs/super.c
do_thaw_all_callback
-> down_write(&sb->s_umount) [Line 1028]
-> emergency_thaw_bdev [Line 1030]
-> thaw_bdev [526]
-> mutex_lock(&bdev->bd_fsfreeze_mutex) [Line 734]

Code Path P2:
// FILE: linux-5.10/fs/block_dev.c
freeze_bdev
-> mutex_lock(&bdev->bd_fsfreeze_mutex) [Line 556]
-> freeze_super [Line 576]
-> down_write(&sb->s_umount) [Line 517]
(a) Deadlock in SysRq command handling for filesystems

Code Path P1:
// FILE: linux-5.10/drivers/scsi/lpfc/lpfc_nvmet.c
lpfc_nvmet_unsol_fcp_issue_abort
-> spin_lock_irqsave(&ctxp->ctxlock, flags) [Line 3502]
-> spin_lock(&phba->sli4_hba.abts_nvmet_buf_list_lock) [Line 3504]

Code Path P2:
// FILE: linux-5.10/drivers/scsi/lpfc/lpfc_nvmet.c
lpfc_sli4_nvmet_xri_aborted
-> spin_lock(&phba->sli4_hba.abts_nvmet_buf_list_lock) [Line 1787]
-> spin_lock(&ctxp->ctxlock) [1794]
(b) Deadlock in the LPFC SCSI driver
```

Figure 12: Two real deadlocks found by DLOS in Linux 5.10.

5.4 Comparison Experiment

We aim to experimentally compare to RacerX [19], which is the sole existing static approach to systematically detect deadlocks in OS kernels. However, RacerX is not open-source, and Linux kernel 2.5.62 checked in its paper is too old to be normally compiled by Clang. Thus, we have to try our best to implement a RacerX-like tool according to its paper.

RacerX [19] performs code analysis with summary caches, which seems similar to function summaries used in our lock-usage analysis. However, RacerX lacks the other two key techniques used in DLOS, namely the reachability-based comparison method to improve the efficiency of locking-cycle detection, and the two-dimensional filtering strategy to drop false positives. To validate the value of these two techniques in comparison, we implement three tools by modifying DLOS: (1) $DLOS_{reach}$ that uses the traditional comparison method in RacerX for locking-cycle detection to replace the reachability-based comparison method in DLOS; (2) $DLOS_{filter}$ that removes the two-dimensional filtering strategy in DLOS; (3) *RacerX-like* that both uses the traditional comparison method for locking-cycle detection and removes the two-dimensional filtering strategy in DLOS.

We run these three tools to check the whole Linux 5.10 code, but the $DLOS_{reach}$ and *RacerX-like* tools run for over 60 hours, without finishing their detection. Thus, for more clear comparison, we select six kernel modules in Linux 5.10, and run these tools and DLOS to check the source code of these kernel modules. These six kernel modules include: two ones (*sb* and *lpfc*) that has real deadlocks found by DLOS, two ones (*fpga* and *ocfs2*) that has false deadlocks found by DLOS, and two ones (*jfs* and *bcache*) that has no deadlock found by DLOS. Table 3 shows the results, and we find that:

	Description	DLOS _{reach}	DLOS _{filter}	RacerX-like	DLOS
<i>sb</i>	Found bugs (real/all)	6/6	6/14	6/14	6/6
	Time usage	30s	14s	27s	16s
<i>lpfc</i>	Found bugs (real/all)	7/7	7/25	7/25	7/7
	Time usage	524s	162s	501s	181s
<i>fpga</i>	Found bugs (real/all)	0/2	0/5	0/5	0/2
	Time usage	21s	9s	18s	11s
<i>ocfs2</i>	Found bugs (real/all)	0/2	0/10	0/10	0/2
	Time usage	936s	214s	892s	253s
<i>jfs</i>	Found bugs (real/all)	0/0	0/0	0/0	0/0
	Time usage	305s	101s	280s	122s
<i>bcache</i>	Found bugs (real/all)	0/0	0/3	0/3	0/0
	Time usage	78s	28s	71s	33s

Table 3: Comparison results of six Linux kernel modules.

First, the DLOS_{reach} tool achieves the same accuracy with DLOS, but it spends more time on locking-cycle detection. Thus, our reachability-based comparison method is more efficient than the traditional comparison method in RacerX, when performing locking-cycle detection.

Second, the DLOS_{filter} tool reports many more false bugs than DLOS, though it finds the real deadlocks found by DLOS. Thus, our two-dimensional filtering strategy is useful to dropping false positives in deadlock detection. Moreover, we observe that the DLOS_{filter} tool spends less time than DLOS. Indeed, without validating the feasibility or concurrency of target code paths, the DLOS_{filter} tool can decrease time usage; but this tool extracts many infeasible target code paths for locking-constraint comparison, which also increases the time usage of locking-cycle detection. As a whole, the decreased time usage is more than the increased time usage in the experiment, and thus the DLOS_{filter} tool has less time usage than DLOS.

Finally, the RacerX-like tool spends less time than the DLOS_{reach} tool, because it does not validate the feasibility or concurrency of target code paths, but causing more false bugs to be reported. The RacerX-like tool spends more time than the DLOS_{filter} tool, because it detects locking cycles with the traditional comparison method, which is less efficient than our reachability-based comparison method; but it achieves the same accuracy with the DLOS_{filter} tool, because neither of them drops false positives. Compared to the RacerX-like tool, DLOS achieves better accuracy in deadlock detection with less time usage.

6 Discussion

Interleaving model. DLOS identifies each target code path from the entry of each function in OS code, and then it considers that two different target code paths identified by our lock-usage analysis can be concurrently executed. To reduce false positives, DLOS validates their concurrency by checking common locks and call graphs using our two-dimensional filtering strategy. As this strategy is infeasible in handling the case that a target code path is concurrently executed with itself, DLOS cannot detect deadlocks occurring in this case.

Detecting deadlocks in other OS kernels. Besides the Linux kernel, DLOS can also check other OS kernels to detect their deadlocks. However, doing so has some practical difficulties. For example, some APIs used in DLOS have different usages between these Oses and Linux, and these Oses have different processes of kernel-code compilation from Linux. At present, we have preliminarily run DLOS in NetBSD to check its kernel source code, and found one real deadlock in the *sysmon* kernel module without false positive. This deadlock has been confirmed by NetBSD kernel developers.

Detecting deadlocks involving waiting queues. Besides the deadlocks caused by locking cycles, incorrect operations on waiting queues can also cause deadlocks in OS kernels. For example, one thread waits for the event E_1 and then triggers the event E_2 , while the other concurrent thread waits for the event E_2 and then triggers the event E_1 , so a deadlock can occur for these two threads. In kernel code, waiting queues and locks can be used together to cause deadlocks, which are more difficult to detect. At present, no static approach (including RacerX) can detect such deadlocks, and thus we plan to extend DLOS to detecting them.

Detecting other locking issues. We believe that DLOS can be extended to detecting other locking issues, such as double locks and using sleep-able locks while holding spinlocks. Indeed, computing locksets and validating code-path feasibility are two important steps in detecting locking issues, and our lock-usage analysis and filtering strategy can effectively perform these steps, respectively.

Limitations and future works. DLOS can be strengthened in some aspects. First, DLOS does not handle function-pointer calls in its lock-usage analysis, and thus it may miss deadlocks involving the code that is reached through function-pointer calls, especially the deadlocks across kernel modules connected by function pointers. To relieve this limitation, we plan to apply existing function-pointer analysis [3, 36] in DLOS to detect more deadlocks and reduce false negatives. Second, to reduce the complexity of analyzing loops and recursive calls, DLOS unrolls each loop and recursive call just once, causing soundness loss in static analysis. Such soundness loss can introduce both false positives and negatives, when DLOS analyzes the code involving loops and recursive calls. To relieve this limitation, we plan to adapt existing loop-oriented analysis [35, 48] in DLOS to soundly handle loops and recursive calls. Thirdly, DLOS does not handle some special cases at present, such as RCU locks, memory barriers, assembly instructions and concurrent execution of the same code path, which may also cause false positives and negatives in deadlock detection. To relieve this limitation, we plan to consider these special cases in our static analysis, to further improve analysis accuracy. Finally, we plan to port DLOS to detecting deadlocks in other OS kernels, and to extend DLOS to detecting deadlocks involving waiting queues as well as other locking issues.

7 Related Work

7.1 Dynamic Analysis of Deadlocks

Many approaches [5, 9–11, 15, 21, 27, 28, 31, 33, 44, 45, 55] dynamically monitor thread execution and lock-related operations to detect locking cycles. Most of them are used for user-level applications. For example, Pulse [31] is an operating system mechanism to detect deadlocks in applications. It periodically identifies long-sleeping application processes and the events they are waiting for, then uses high-level speculative execution to a general resource graph about each identified application process, and finally detects cycles in the graph as deadlocks. UnDead [55] is an efficient dynamic approach for deadlock detection. It uses several techniques to reduce runtime overhead, such as only recording unique lock dependencies (identical to locking constraints in this paper) for every thread during the execution and dropping unnecessary information in runtime recording.

Lockdep [33] is a kernel lock-usage validator, which can find different kinds of lock-related bugs, such as double locks and deadlocks. Lockdep performs runtime monitoring and checking based on the granularity of lock class, which describes a group of locks that are logically the same with respect to locking rules. Specifically, Lockdep dynamically tracks the state of each lock class and checks the dependencies between different lock classes. If any state or dependency is incorrect when lock-related operations are performed, Lockdep will report related bugs at runtime.

By using exact runtime information about thread execution and lock-related operations, dynamic analysis approaches can effectively reduce false positives in deadlock detection. However, dynamic analysis requires substantial test cases to achieve high testing coverage and reduce false negatives, and it also introduces runtime overhead for the tested programs.

7.2 Static Analysis of Deadlocks

Some approaches [30, 41, 42, 46, 52] use static analysis to detect deadlocks in user-level applications, without actually running the applications. Naik et al. [30] design a sound static approach to check deadlocks in C programs. For the checked program, this approach performs context-sensitive and thread-sensitive analysis on its inter-procedural control flows, based on abstract interpretation. During the analysis, this approach checks lock-related operations to extract lock dependencies and detect locking cycles as possible deadlocks. This approach also uses a non-concurrency analysis to drop false positives, by checking common locks and thread-creation/joining operations. Santhiar et al. [46] design a static approach to detect deadlocks in asynchronous C# programs. This approach uses a new representation of the mixed synchronous and asynchronous control flows, and constructs a deadlock detection graph based on this representation. However, OS kernels and

user-level applications have different concurrency models (described in Section 2.2), and thus these approaches are ineffective in detecting deadlocks in OS kernels.

To our knowledge, RacerX [19] is the sole existing static analysis approach to systematically detect deadlocks in OS kernels. However, it has a high false positive rate of 46%, due to neglecting the feasibility and concurrency of code paths; and its locking-cycle detection method is simple and inefficient. Compared to RacerX, DLOS checks the feasibility and concurrency of code paths to achieve better accuracy, and uses a reachability-based comparison method in locking-cycle detection to achieve higher efficiency. Besides, we also note that Breuer et al. [6–8] have several works that focus on detecting deadlocks caused by sleeping while holding spinlocks. As these works have no systematic technique of detecting deadlocks caused by locking cycles, we do not particularly introduce and compare to these works in this paper.

7.3 Detection of Kernel Concurrency Bugs

Besides deadlocks, OS kernels also suffer from other kinds of concurrency bugs, such as data races and atomicity violations. To detect these kernel concurrency bugs, some approaches [2, 12, 18, 47] use static or dynamic lockset analysis to track shared variables and lock-related operations, and some approaches [20, 25, 29] perform sampling to monitor concurrent memory accesses. Moreover, to actually cover infrequent thread interleavings and detect hard-to-find concurrency bugs, some approaches [22–24, 26, 53] perform random thread scheduling or coverage-guided thread-interleaving exploration in runtime testing. Though these approaches do not target deadlocks, some of their techniques (like lockset analysis) are useful for DLOS in deadlock detection.

8 Conclusion

Deadlocks in OS kernels are dangerous and hard-to-find. To detect these bugs, we design a practical static analysis approach named DLOS. It has three key techniques, including a summary-based lock-usage analysis to efficiently extract code paths containing distinct locking constraints, a reachability-based comparison method to efficiently detect locking cycles, and a two-dimensional filtering strategy to effectively drop false positives. In the evaluation, DLOS finds 54 real deadlocks in Linux 5.10, and 31 of them have been confirmed.

Acknowledgment

We thank our shepherd and anonymous reviewers for their helpful advice on the paper. We also thank Linux kernel developers, who gave useful feedback and advice to us. This work was supported by the National Natural Science Foundation of China under Project 62002195.

References

- [1] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.
- [2] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 255–268, 2019.
- [3] Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. Effective detection of sleep-in-atomic-context bugs in the Linux kernel. *ACM Transactions on Computer Systems (TOCS)*, 36(4):1–30, 2020.
- [4] Jia-Ju Bai, Yu-Ping Wang, and Shi-Min Hu. AutoPA: automatically generating active driver from original passive driver code. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO)*, pages 288–299, 2018.
- [5] Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Proceedings of the 2005 Haifa Verification Conference*, pages 208–223, 2005.
- [6] Peter T Breuer and Simon Pickin. Checking for deadlock, double-free and other abuses in the Linux kernel source code. In *Proceedings of the 2006 International Conference on Computational Science*, pages 765–772, 2006.
- [7] Peter T Breuer, Simon Pickin, and Maria Larrondo Petrie. Detecting deadlock, double-free and other abuses in a million lines of Linux kernel source. In *Proceedings of the 30th NASA Software Engineering Workshop*, pages 223–233, 2006.
- [8] Peter T Breuer and Marisol Garcíá Valls. Static deadlock detection in the Linux kernel. In *Proceedings of the 9th Ada-Europe International Conference on Reliable Software Technologies*, pages 52–64, 2004.
- [9] Yan Cai and WK Chan. MagicFuzzer: scalable deadlock detection for large-scale applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 606–616, 2012.
- [10] Yan Cai and Qiong Lu. Dynamic testing for deadlocks via constraints. *IEEE Transactions on Software Engineering (TSE)*, 42(9):825–842, 2016.
- [11] Yan Cai, Ruijie Meng, and Jens Palsberg. Low-overhead deadlock prediction. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, pages 1298–1309, 2020.
- [12] Qiu-Liang Chen, Jia-Ju Bai, Zu-Ming Jiang, Julia Lawall, and Shi-Min Hu. Detecting data races caused by inconsistent lock protection in device drivers. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 366–376, 2019.
- [13] Clang: a LLVM-based compiler for C/C++ program. <https://clang.llvm.org/>.
- [14] CLOC: count lines of code. <https://cloc.sourceforge.net>.
- [15] Tiago Cogumbreiro, Raymond Hu, Francisco Martins, and Nobuko Yoshida. Dynamic deadlock verification for general barrier synchronisation. In *Proceedings of the 20th International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 150–160, 2015.
- [16] Linux commit 57ba4cb85bff. <https://github.com/torvalds/linux/commit/57ba4cb85bff>.
- [17] Linux commit 01d01caf19ff. <https://github.com/torvalds/linux/commit/01d01caf19ff>.
- [18] Pantazis Deligiannis, Alastair F Donaldson, and Zvonimir Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers. In *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*, pages 166–177, 2015.
- [19] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th International Symposium on Operating Systems Principles (SOSP)*, pages 237–252, 2003.
- [20] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th International Conference on Operating Systems Design and Implementation (OSDI)*, pages 151–162, 2010.
- [21] Mahdi Eslamimehr and Jens Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE)*, pages 353–365, 2014.
- [22] Pedro Fonseca, Rodrigo Rodrigues, and Björn B Brandenburg. SKI: exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th International Conference on Operating Systems Design and Implementation (OSDI)*, pages 415–431, 2014.

- [23] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the 28th International Symposium on Operating Systems Principles (SOSP)*, pages 66–83, 2021.
- [24] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzler: finding kernel race bugs through fuzzing. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, pages 754–768, 2019.
- [25] Yunyun Jiang, Yi Yang, Tian Xiao, Tianwei Sheng, and Wenguang Chen. DRDDR: a lightweight method to detect data races in Linux kernel. *Journal of Supercomputing*, 72(4):1645–1659, 2016.
- [26] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS)*, 2022.
- [27] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering (FSE)*, pages 327–336, 2010.
- [28] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 30th International Conference on Programming Language Design and Implementation (PLDI)*, pages 110–120, 2009.
- [29] KCSAN: concurrency sanitizer for the Linux kernel. <https://github.com/google/ktsan/wiki/KCSAN>.
- [30] Daniel Kroening, Daniel Poetzl, Peter Schrammel, and Björn Wachter. Sound static deadlock analysis for C/Pthreads. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, pages 379–390, 2016.
- [31] Tong Li, Carla Schlatter Ellis, Alvin R Lebeck, and Daniel J Sorin. Pulse: a dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, pages 31–44, 2005.
- [32] LLVM compiler infrastructure. <https://llvm.org/>.
- [33] Lockdep: runtime locking correctness validator in the Linux kernel. <https://www.kernel.org/doc/html/latest/locking/lockdep-design.html>.
- [34] Linux kernel locking documents. <https://www.kernel.org/doc/html/latest/locking/index.html>.
- [35] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of the 2009 International Symposium on Code Generation and Optimization (CGO)*, pages 136–146, 2009.
- [36] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 26th International Conference on Computer and Communications Security (CCS)*, pages 1867–1881, 2019.
- [37] Lanyue Lu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the 11th International Conference on File and Storage Technologies (FAST)*, pages 31–44, 2013.
- [38] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–339, 2008.
- [39] Ivan Matosevic and Tarek S Abdelrahman. Efficient bottom-up heap analysis for symbolic path-based data access summaries. In *Proceedings of the 2012 International Symposium on Code Generation and Optimization (CGO)*, pages 252–263, 2012.
- [40] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. Scalable and incremental software bug detection. In *Proceedings of the 9th International Symposium on Foundations of Software Engineering (FSE)*, pages 554–564, 2013.
- [41] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 386–396, 2009.
- [42] Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent Go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*, pages 174–184, 2016.
- [43] Leonid Ryzhyk, Yanjin Zhu, and Gernot Heiser. The case for active device drivers. In *Proceedings of the 1st Asia-Pacific Workshop on Systems (APSys)*, pages 25–30, 2010.

- [44] Malavika Samak and Murali Krishna Ramanathan. Multithreaded test synthesis for deadlock detection. In *Proceedings of the 2014 International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 473–489, 2014.
- [45] Malavika Samak and Murali Krishna Ramanathan. Trace driven dynamic deadlock detection and reproduction. In *Proceedings of the 19th International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 29–42, 2014.
- [46] Anirudh Santhiar and Aditya Kanade. Static deadlock detection for asynchronous C# programs. In *Proceedings of the 38th International Conference on Programming Language Design and Implementation (PLDI)*, pages 292–305, 2017.
- [47] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.
- [48] Yulei Sui, Xiaokang Fan, Hao Zhou, and Jingling Xue. Loop-oriented pointer analysis for automatic simd vectorization. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):1–31, 2018.
- [49] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th International Symposium on Operating Systems Principles (SOSP)*, pages 207–222, 2003.
- [50] Linux kernel SysRq documents. <https://www.kernel.org/doc/html/latest/admin-guide/sysrq.html>.
- [51] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.
- [52] Amy Williams, William Thies, and Michael D Ernst. Static deadlock detection for Java libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*, pages 602–629, 2005.
- [53] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. KRACE: data race fuzzing for kernel file systems. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, pages 1643–1660, 2020.
- [54] Z3: a theorem prover. <https://github.com/Z3Prover/z3>.
- [55] Jinpeng Zhou, Sam Silvestro, Hongyu Liu, Yan Cai, and Tongping Liu. Undead: detecting and preventing deadlocks in production software. In *Proceedings of the 32nd International Conference on Automated Software Engineering (ASE)*, pages 729–740, 2017.



Modulo: Finding Convergence Failure Bugs in Distributed Systems with Divergence Resync Models

Beom Heyn Kim^{§†}, Taesoo Kim^{§‡}, and David Lie[†]

[§]Samsung Research [†]University of Toronto [‡]Georgia Institute of Technology
{beomhey.n.kim,tsgates.kim}@samsung.com, lie@eecg.toronto.edu

Abstract

While there exist many consistency models for distributed systems, most of those models seek to provide the basic guarantee of convergence: given enough time and no further inputs, all replicas in the system should eventually converge to the same state. However, because of *Convergence Failure Bugs* (CFBs), many distributed systems do not provide even this basic guarantee. The violation of the convergence property can be crucial to safety-critical applications collectively working together with a shared distributed system. Indeed, many CFBs are reported as major issues by developers. Our key insight is that CFBs are caused by divergence, or differences between the state of replicas, and that a focused exploration of divergence states can reveal bugs in the convergence logic of real distributed systems while avoiding state explosion. Based on this insight, we have designed and implemented Modulo, the first Model-Based Testing tool using *Divergence Resync Models* (DRMs) to systematically explore divergence and convergence in real distributed systems. Modulo uses DRMs to explore an abstract state machine of the system and derive schedules, the intermediate representation of test cases, which are then translated into test inputs and injected into systems under test (SUTs). We ran Modulo to check ZooKeeper, MongoDB, and Redis and found 11 bugs (including 6 previously unknown ones)

1 Introduction

The emergence of cloud-scale applications has driven the need for distributed storage systems to support them. To provide availability and scalability, those systems replicate data across several replicas, which may be distributed in a single datacenter or even globally across several datacenters [1–5, 32, 39, 50, 57, 62]. To tolerate network partitions and delays, many of these systems adopt weaker-consistency guarantees [58, 59], allowing them to replicate data asynchronously. This means that clients connected to different replicas may observe different states of the data. The exact order and delay of concurrent operations on replicated data is

governed by a “consistency model,” which attempts to strike a balance between intuitive behavior (favoring stronger consistency guarantees) and scalability and partition tolerance (favoring weaker guarantees).

Regardless of these differences, most of consistency models in practice guarantee a common property, which is that given enough time and no further modifications to the data, all replicas will eventually arrive at the same contents for the data—something we call the *convergence guarantee* also known as eventual consistency. However, distributed systems are inherently designed to be *temporarily inconsistent* so that they may continue to respond to requests, even as they *converge* to a consistent state by replicating data among the replicas. We call this temporary inconsistency *divergence*. Yet, divergence can cause more than temporary inconsistency in the presence of failures. Divergence in the presence of failures may also lead to *conflicts*, where different replicas have incompatible states, which can only be resolved by truncating or removing data from one or more of the replicas. While it is not the sole cause, we find that a major cause of systems failing to converge is defects in the convergence logic after the failure recovery. Such bugs leading to convergence failures are named *Convergence Failure Bugs* (CFBs).

Looking into bug databases of a couple of systems for the period from 2010 to 2017, we found about 10 bugs that are already reported and fixed by developers and external users [23, 31, 33, 34, 51, 55, 56, 60, 64]. They are all marked by developers as either “Blocker”, “Critical” or “Major” in terms of severity. This demonstrates that CFBs are perceived by developers as real, prevalent and important bugs to find and fix. In some case, the convergence failure is noticed by developers as visible to clients [31], which can directly cause clients to make incorrect decisions leading to serious consequences. Thus, a convergence failure can be crucial to safety-critical applications collectively working together through a shared distributed system.

To exercise distributed systems’ convergence logic, more divergence than would naturally occur during regular use needs to be generated. This paper presents *Modulo*, the first Model-Based Testing tool that systematically explores differ-

ent divergence states by alternately injecting events that cause divergence and convergence into the real distributed systems.

Modulo overcomes limitations in previous solutions for detecting CFBs in distributed systems. On one hand, distributed systems model-checkers [27, 29, 35, 41, 42, 44, 52, 61] aim to provide formal verification of a distributed system, and as such must ensure that they exhaustively explore the state space of the system under test (SUT). To achieve exhaustive exploration, they must tightly control all nondeterministic events so as to drive the SUT through all possible states. Unfortunately, tightly controlling all events leads to the well-known “state-explosion problem,” as the number of states grows exponentially with the number of events that are controlled. To reduce the severity of state-explosion, a smart and insightful abstraction of target behavior is needed. None of existing model-checkers has explored the state-space consisting of divergence and convergence.

On the other hand, random testing approaches, such as Jepsen [38], do not aim for formal verification but rather simply to find bugs, and thus they need not exhaustively explore all states. This frees them from having to control all nondeterministic events. Instead, random testing approaches inject a targeted set of external events (randomly of course) and, rather than controlling all other events, allow the SUT to randomly visit states depending on how events interleave naturally during execution. Random testing approaches typically do not record the states explored, so they cannot provide any guarantee or measure of state-space coverage. Moreover, because they do not know which events are important, they may not be able to provide a sequence of inputs that can reliably reproduce the bug.

Modulo’s key contributions stem from our observation that *many CFBs arise from flaws in the convergence logic of distributed systems, and are orthogonal to other functions of the system*. Thus, CFBs can often be reproduced purely by partially controlling only the few events that lead to convergence and divergence. This partial control allows Modulo to significantly reduce the severity of the state explosion problem, while still enabling it to deeply explore different divergence states of the SUT.

Different distributed systems have different techniques to converge replicas after a failure. To abstract these differences so that it can generalize across different systems, Modulo introduces *Divergence Resync Models* (DRMs), which consist of an *Abstract Execution Model* (AEM) and a *Concrete Execution Model* (CEM). The AEM describes abstract conditions for convergence and divergence events. For example, systems like ZooKeeper and MongoDB require quorum before they can accept client requests that could cause divergence between replicas. The AEM for these systems thus specifies the conditions under which the systems quorum will have been achieved (i.e., the majority of replicas are available). The CEM maps the AEM conditions, as well as divergence and convergence events, to API calls for a specific SUT. Mod-

ulo uses the AEM to generate schedules of abstract events that alternate between divergence and convergence and the CEM to execute these schedules on the SUT to search the convergence code of the system for CFBs.

We ran Modulo on ZooKeeper, MongoDB, and Redis as SUTs and found 11 CFBs, including 6 new ones that had not been found before. For each of these bugs, Modulo provides a schedule of inputs that deterministically triggers the bug. To find these CFBs, we used 5 DRMs—1 for ZooKeeper, 1 for MongoDB, and 3 for Redis, which range from 72-782 lines of code in size.

We made the following novel contributions:

- As far as we know, Modulo is the first systematic test generation system, specifically designed to discover CFBs.
- We introduce the concept of Divergence Resync Models (DRMs) that inject events specifically designed to elicit and discover the existence of CFBs.
- We design, implement and evaluate Modulo, a system that uses DRMs to find CFBs in real distributed systems.
- We perform an empirical study to demonstrate the effectiveness of the proposed approach by integrating the prototype to 3 mature open-source distributed systems: ZooKeeper, MongoDB, and Redis. Modulo was able to find several critical CFBs in them.

§2 gives the overview of divergence and convergence, the core concepts of Modulo and DRMs, and provides an example CFB that Modulo can find. §3 describes the architecture of Modulo and the 5 DRMs we use in this study. We then document our experience and empirical results of applying Modulo to mature open-source distributed systems in §4. Subsequently, we present further discussion on Modulo compared to previous proposals in §5 and discuss related work in §6. Finally, we draw our conclusions in §7.

2 Overview

2.1 System Model

We model a distributed storage system as a set of replicas, each of which is a key-value store. In an idealized system, the storage system consists of one replica, and each write would transition the key-value store in the system from one state to the next. However, in the distributed implementation of the system, each write is applied to one of the replicas, and then the distributed storage system asynchronously replicates the write to the remaining replicas until every replica converges to the same state. Thus, at any given time, there can be several replicas that differ from our idealized single replica. Our target system should manage updates and resolve conflicts to maintain the convergence of replicas. That is, our proposed

approach is not designed to test those systems allowing multi-master updates with gossip protocols where the convergence property is not guaranteed.

Our approach injects failures into the system-under-test to drive it into corner cases. Failures are erroneous states where synchronization between replicas is interrupted, so convergence does not occur without reverting to the normal state via failure recoveries. Failures may be caused by several reasons, such as crash of replicas, suspension of replicas, and links failures between replicas. After recovering from failures, convergence should occur and replicas must have identical structure in terms of write sequences contained in their log. Regarding the convergence property, we focus on those systems that are partially synchronous, although there is nothing stopping us from applying Modulo to eventually consistent systems such as Cassandra, which is left as a future work. Quorum-based systems we tested with Modulo require a quorum to elect a leader, but may ingest writes expecting failures will be recovered soon. In contrast, Redis does not use leader election, so it does not require a quorum to start servicing clients. It lets the sync source to replicate any changes down to the sync targets recursively.

2.2 Divergence and Convergence

We formally define *divergence* as the total number of writes that need to be applied across all replicas to make their key-value stores equal to the single idealized replica. *Convergence* is simply the complement of divergence (i.e., $convergence = -divergence$), where the replicas are said to be converged when divergence is zero. *Resynchronization* (resync) is an operation implemented by distributed storage systems to achieve convergence after a failed replica recovers. Resync reduces divergence by replicating writes from a replica to the recovered replica and may implement *conflict resolution* to eliminate write operations that prevent the replicas from achieving convergence. Conflict resolution is particularly complex as it usually results in data loss, which storage systems seek to avoid unless absolutely necessary.

Divergence occurs in the natural course of the operation of a distributed system as writes are applied to replicas. However, under normal circumstances, the amount of divergence is usually small, as systems aim to replicate writes fairly quickly, subject to standard networking and processing delays. However, failures may further increase divergence. For instance, replica crashes and network outages can prevent replicas from replicating operations for an extended period of time. This will trigger defects related to assumptions about the resources needed to track outstanding operations or about the length of time that replicas may be unavailable. Repeated failures may result in replicas changing leader or master roles, which can result in conflicts, allowing Modulo to exercise the conflict resolution logic of distributed systems.

A simple example illustrating the concept of divergence

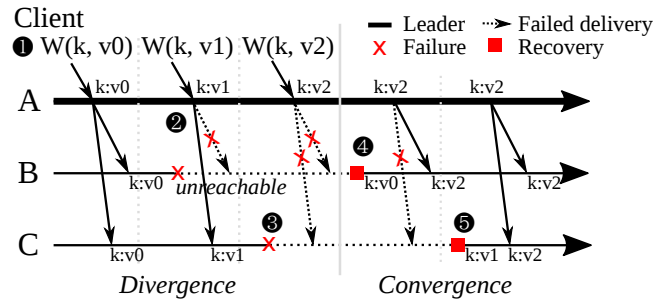


Figure 1: **Divergence and convergence.** Replicas A, B and C diverge their states upon failures of B and C. Since the leader replica A is alive, the latest state of A can be replicated to replicas B and C upon their recovery, resulting in the state *convergence*.

and convergence is given in Figure 1. The divergence example starts in an initial state where A is the leader replica and B and C are non-leader replicas. At time 1, a write of v_0 to k , $W(k, v_0)$, is sent by a client to A, A applies the operation and then replicates it to B and C. At time 2, B fails and, another write, $W(k, v_1)$, is sent to A. Only A and C can apply the write, thus there is some amount of temporary divergence among replicas. At time 3, C fails, and another write, $W(k, v_2)$, is sent to A. Only A can apply the write, resulting in even more divergence among the 3 replicas. Convergence can simply be regarded as the decrease of divergence in the system. At time 4, B recovers from the failure and rejoins the distributed system. Most distributed systems implement resync procedures that attempt to bring B up to date with the most recent state on the other replicas. Thus, B can converge with A by replicating $W(k, v_2)$. At time 5, C recovers and resyncs with A by replicating $W(k, v_2)$, resulting in full convergence among the 3 replicas. Now, there is no divergence in the system. (i.e., all replicas have the same key-value stores).

2.3 An Example Bug

To illustrate the type of bugs Modulo will find, we give an example of a new bug that Modulo discovered in ZooKeeper version 3.4.11 [8]. ZooKeeper requires a quorum of replicas to be online to operate, where quorum is defined as more than one-half the total number of replicas. In each epoch, which is a predefined period of time, the replicas in the quorum elect one of them as the “leader.” All write operations are serialized through the elected leader, and all other replicas (called “followers”) replicate operations from the leader.

ZooKeeper replicas employ 2 mechanisms to save data so that it can be recovered after a crash. First, the replicas use a write-ahead transaction log that can be replayed after a failure to recover the state of data that did not properly persist. Transactions are appended into the log as they get committed. Because all changes caused by transactions are

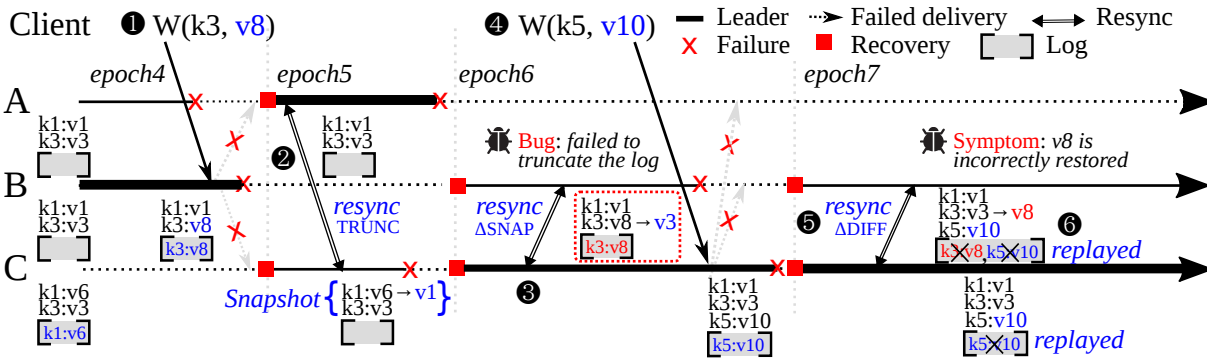


Figure 2: **Running example (ZooKeeper Bug#1)**. The replica B fails to truncate the past log ($k3: v8$) upon recovery with the SNAP resync (see, §2.3). To observe this bug, another failure and recovery step is required right after the bug is triggered (6). Modulo successfully formulated the exact sequences of steps to trigger this previously unknown CFB.

sequentially logged, replaying transactions in the log will restore the state of a replica. Second, replicas periodically clear the transaction log and persist a snapshot of in-memory data to disk, which can then be reloaded into memory after a failure. Taking a snapshot of memory is considerably slower than writing transactions into the write-ahead log as they occur, so ZooKeeper only takes snapshots after the transaction log has grown to some point or after resync following a failure depending on the resync logic. For example, in ZooKeeper 3.4.11, taking snapshot occurs on followers if they resync with a new leader via SNAP or by sending a truncation request, which are further explained in following paragraphs. The example is replica C taking a snapshot after the resync at time 2 in Figure 2.

ZooKeeper uses 2 mechanisms to resync replicas after replicas have recovered from a failure. One mechanism is DIFF resync, where another replica transfers all missing operations from its transaction log to the recovered replica. The other mechanism is SNAP resync, where another replica sends its entire key-value store to the recovered replica. ZooKeeper picks DIFF resync if the leader’s log contains all transactions required for resync. However, this can lead to problems as old log entries may be purged by an earlier snapshot. If ZooKeeper’s resync logic determines the leader does not have all required transactions in its log, then the SNAP mechanism will be selected. For example, in ZooKeeper version 3.4.11, when followers resync with the leader that does not have a log containing entries that are newer than its snapshot, the SNAP mechanism is used (e.g., SNAP resync at time 3 in Figure 2).

The base case that can happen during the resync is replicating operations that have not been replicated to those replicas recovered from a failure. Moreover, it may be necessary for a recovering replica to *truncate* its local write-ahead log and remove conflicting operations. During resync, a leader sends a truncation request (TRUNC) to a follower. Conflicting operations may exist if one replica had been the leader and committed some operations that had not yet been replicated

to other replicas before failing, and another replica is subsequently elected as the new leader and commits another set of operations, resulting in two conflicting sets of operations.

We illustrate the example bug in Figure 2. Suppose we initially have replicas A, B and C and the epoch is at 4. Both A and B have the same set of key-value pairs $k1: v1, k2: v2, k3: v3, k4: v4$ and $k5: v5$. Also, their log contains entries for all writes creating the key-value set. C has the same key-value set except for $k1: v6$ and its log additionally contains the entry for the write, $W(k1, v6)$. Currently, A and B are up and C is down. B is the leader of the epoch 4.

At time 1, A crashes and a write, $W(k3, v8)$, is made, which B accepts and commits¹. Then, B crashes. At time 2, A and C restart and resync using TRUNC. C restores its key-value set by replaying its truncated log and takes a snapshot. Note that neither A nor C has seen the write, $W(k3, v8)$, yet at this point. Then, A and C crash and B and C restart. C becomes the new leader. At time 3, because C has no log entry newer than its snapshot, C resync with B using the SNAP mechanism. Yet, C does not send a truncation request to B, so B accepts and restores using the C’s snapshot but fails to truncate the write, $W(k3, v8)$, from its log, which is the root cause of the bug. However, at this point, the bug is not apparent yet. At time 4, B crashes and another write, $W(k5, v10)$, is committed on C. Then, C crashes and B and C restart. Now, C becomes the leader. At time 5, B and C use DIFF resync to replicate $W(k5, v10)$ from C to B. At time 6, B replays its log and restores $k3: v8$, while C still has $k3: v3$. Because of the failure to truncate the log entry for $k3: v8$ on B, the replicas believe they have converged when in fact they have not, and the system fails to reach convergence.

This example illustrates several key features of the CFBs that Modulo is designed to discover. First, a long series of very specific steps is required to trigger the bug—far larger than are likely to happen simply due to randomized stress testing. The series of steps is essentially the alternating sequence of

¹ $W(k2, v7)$ injected at the epoch 3 and $W(k4, v9)$ injected at the epoch 5 are not shown in the figure.

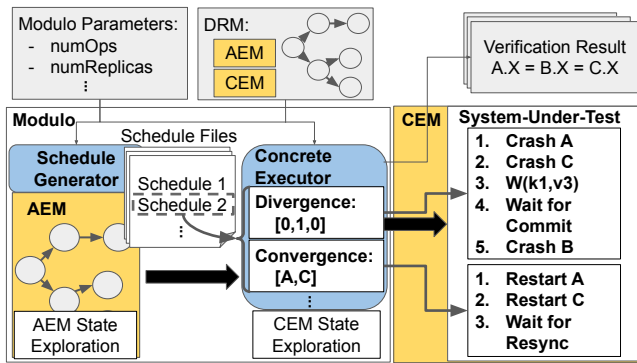


Figure 3: Modulo Architecture. Gray boxes are input and output of Modulo. Blue boxes are Modulo components. Yellow boxes are DRM components.

convergence and divergence that causes the CFB to surface. Modulo specifically targets the generation of such sequences using DRMs, which we discuss in Section 3. Second, because the sequence is very deep, it would be very difficult to find such bugs through a naïve search of the entire state space of all the replicas. However, the root cause and nature of the bug make it orthogonal to the detailed internal state of the replicas, which can be influenced by internal events such as thread interleaving and the order of lock acquisitions. Instead, it is the generation of divergence and convergence events between the replicas in ZooKeeper that triggers the bugs, which are the events that Modulo seeks to explore. By focusing only on controlling events related to divergence and convergence, such as replica failure and recovery, Modulo is able to explore deep sequences such as these without being constrained by state explosion.

3 Modulo

As shown in Figure 3, Modulo consists of 2 core components: a schedule generator and a concrete executor. To use Modulo, the user provides a DRM and 2 parameters: *numReplicas*, which indicates the number of replicas, and *numOps*, which indicates the total number of writes that will be applied to the SUT. Modulo then uses the DRM to produce a schedule of divergence transitions ($\mathcal{D} \rightarrow$) and convergence transitions ($\mathcal{C} \rightarrow$), which are then executed by the concrete executor on the SUT. $\mathcal{D} \rightarrow$ cause more divergence in a system while $\mathcal{C} \rightarrow$ cause SUT convergence among available replicas. After executing each schedule, the concrete executor waits for the SUT to be quiescent after pre-configured time duration. Then, it checks whether all replicas in the SUT have converged (i.e., have identical state) by reading values of each key and compare those across replicas. Finally, the result is recorded in a file for more detailed analysis.

A DRM contains 2 subcomponents: an Abstract Execution Model (AEM) and a Concrete Execution Model (CEM). The

AEM describes the conditions under which $\mathcal{C} \rightarrow$ and $\mathcal{D} \rightarrow$ may take place. For example, an SUT may require a quorum of replicas to be available before write operations will be accepted and divergence may occur, which would be specified in the AEM. AEMs describe such requirements abstractly in terms of the AEM state, given in Table 1, such that a single AEM may be used to test multiple SUTs. For example, both ZooKeeper and MongoDB require quorums, so an AEM that models quorums is used to test both, while an AEM that does not model quorums may be needed to test systems that do not rely on quorum. The output of the schedule generator in Modulo is a set of schedule files, which are consumed by the concrete executor. We describe this further in §3.3. The concrete executor uses the CEMs to map $\mathcal{C} \rightarrow$ and $\mathcal{D} \rightarrow$ in the schedules to concrete operations that drive a SUT into the states dictated by the schedule. As such, CEMs are necessarily specific to the SUT.

An AEM specifies a finite state machine of the system-under-test and the CEM translates transitions in the AEM into corresponding transitions on the system-under-test. The user is required to abstract away unnecessary details in the AEM to limit the state space being tested to interesting states. The relationship between the AEM and the CEM can thus be viewed as the AEM specifying the state space to test and the CEM translating tests specified by that state space into concrete tests to run on the actual SUT. Modulo exhaustively explores the AEM’s finite state machine. Failures are detected directly on the SUT when it fails to converge after a certain amount of time. There are differences in the DRMs to test each SUT differently. For instance, Redis’s DRMs need a way to model the link failures and recoveries between pairs of replicas, different from ZooKeeper’s and MongoDB’s DRM.

We describe the most generic form of AEMs and CEMs below, which can be flexibly extended to more sophisticated models. Table 1 and 2 are for the baseline AEM used for ZooKeeper and MongoDB. Our baseline AEM abstract away the role of replicas, but the leader is distinguished during concrete execution of CEM to find out to which replica a write should be injected. For more complex AEMs like the one used for Redis, we need to extend states and transitions of AEM to model the network link failures.

3.1 Abstract Execution Model

The AEM specifies a state machine, whose state space Modulo explores to produce the schedule of $\mathcal{C} \rightarrow$ and $\mathcal{D} \rightarrow$. At each state, Modulo systematically performs $\mathcal{C} \rightarrow$ and $\mathcal{D} \rightarrow$ depending on whether the guard conditions specified in the AEM are met or not. The guard conditions are boolean functions over the state of the AEM. During $\mathcal{D} \rightarrow$, Modulo simulates a client that sends zero or more write operations from the sequence of write operations $W : \langle w_n | n \in \{0..numOps\} \rangle$, where *numOps* is the parameter specified as part of the test configuration. The term *replicaState* is a vector of length

Variable	Description
replicaState	List of non-negative integer values. State of all replicas that defines the latest write operation applied
onlineStatus	List of boolean values. Status of the replica indicating if the replica is online or offline

Table 1: AEM State Description.

$numReplicas$, with each element indicating the index of the latest write operation in W that a particular replica has observed. The term $onlineStatus$ is also a vector of length $numReplicas$, which stores the status of each replica as either *online*, meaning that the replica is available, or *offline*, meaning that the replica is unavailable because it has failed for reasons such as a crash or a link failure.

Modulo performs $C \rightarrow$ and $\mathcal{D} \rightarrow$ on the AEM according to the transition descriptions given in Table 2. For $\mathcal{D} \rightarrow$, Modulo fails zero or more replicas, followed by zero or more write operations from the write sequence (though obviously there should be either non-zero replica failures or non-zero writes). For example, consider the replicaState of a 3-replica system, which can be represented by a tuple $[R_A, R_B, R_C]$ with each element corresponding to a replica’s replicaState. Recall that the replicaState is the index of the latest write that the replica has ideally replicated. Thus, a divergence transition that applies a write to replica A, which is replicated to replica B, will change the replicaState from $[1, 1, 1]$, to $[2, 2, 1]$. Because replica C did not replicate the write due to a failure, its replicaState does not increase. For $C \rightarrow$, Modulo returns one or more replicas back to operation and initiates resync. In cases where resync is automatic, the AEM will simply model all online replicas as achieving the latest write index. However, some systems, like Redis, allow manual resync between a subset of replicas, in which case the AEM may explore states with different subsets of replicas resynchronizing. In general, if the type of resync or failure that can occur depends on the abstract AEM state, then the AEM model will specify the type of resync or failure for each $C \rightarrow$ and $\mathcal{D} \rightarrow$ in the generated schedules accordingly. If it does not, then the CEM will run the SUT several times with the same schedule, trying out the different SUT-specific failure and resync methods. The CEM, which we discuss in §3.2, specifies many of the details on how replica failures are caused and how Modulo can tell if resync is complete.

Modulo’s schedule generator applies symmetry reduction [15] to remove schedules that have identical states. For example, in a system with replicaState of $[3, 3, 1]$, where replicas A and B have replicated up to write #3 while replica C has only replicated up to write #1, failing replica A or replica B is symmetrical, so Modulo will only produce one schedule for both of those cases. One notable caveat is that some systems, such as ZooKeeper, distinguish one leader replica from the others. We can extend AEMs in a straightforward way by

Transition	Description
convergence	replicaState: set each online replica’s write index to the latest write that the replica can resynchronize to onlineStatus: set one or more replicas to online
divergence	replicaState: increase replica’s write index based on number of writes applied onlineStatus: set zero or more replicas to offline

Table 2: AEM Transition Description.

adding a state variable to track which replica is the leader, and a leader and the non-leader will be considered not identical for the purposes of symmetry reduction.

An AEM produces schedules for a CEM to interpret and inject events to a SUT. The following is the schedule we used to find our example ZooKeeper bug, which was generated by using Q/C/Z-DRM (see §3.3) with the modulo parameters $numOps = 5$ and $numReplicas = 3$. The example Figure 2 illustrates what happens between ⑤ and ⑩. Note that we use integer values to indicate the degree for divergence but to identify replicas for convergence (i.e., 0 for A, 1 for B and 2 for C).

$\mathcal{D} \rightarrow$: Divergence, $C \rightarrow$: Convergence
① $\mathcal{D} \rightarrow [0, 0, 1]$ // introducing divergence by making C commit a write $W(k1, v6)$ while other replicas are failed; then fail C
② $C \rightarrow [0, 1]$ // introducing convergence by recovering failures of A and B and having them resync; C remains failed
③ $\mathcal{D} \rightarrow [0, 1, 0]$
④ $C \rightarrow [0, 1]$ // epoch 4 begins; B becomes a leader
⑤ $\mathcal{D} \rightarrow [0, 1, 0]$ // $W(k3, v8)$ is committed on B
⑥ $C \rightarrow [0, 2]$ // resync TRUNC; C takes a snapshot
⑦ $\mathcal{D} \rightarrow [1, 0, 0]$ // skipped in the figure
⑧ $C \rightarrow [1, 2]$ // SNAP resync; truncation fails (Bug)
⑨ $\mathcal{D} \rightarrow [0, 0, 1]$ // $W(k5, v10)$ is committed on C
⑩ $C \rightarrow [1, 2]$ // resync DIFF; Bug manifests
⑪ $C \rightarrow [0]$

3.2 Concrete Execution Model

The purpose of a CEM is to translate the abstract $C \rightarrow$ and $\mathcal{D} \rightarrow$ in the schedules generated from an AEM into concrete actions that can be performed on an SUT, to drive it down the individual schedules. For example, a $\mathcal{D} \rightarrow$ may indicate that one of the replicas advances its write index while the others do not, which the CEM may translate as failing 2 replicas and then injecting a write into the SUT. As such, the concrete executor and CEM share some similarities with concrete model-checkers, except that the Modulo’s concrete executor only explores $C \rightarrow$ and $\mathcal{D} \rightarrow$ sequences specified by the schedules generated by the AEM, and thus only control the aspects of the concrete state that map onto the abstract state

of the AEM, namely whether replicas are online or offline, and what writes are replicated by each replica.

In most cases, $C \rightarrow$ and $\mathcal{D} \rightarrow$ can be mapped to a set of SUT-specific APIs to write keys in the SUT and to check if resync has completed. In some cases, CEMs may also need to monitor log files to infer whether the certain aspects of resynchronization, such as leader election, have completed. Finally, in some extreme cases, we may need to instrument the SUT itself to reveal such interfaces to the CEM. For example, if we wanted to make the leader replica explicit when the system does not provide such information, we can replace the default leader election protocol with the one that can explicitly report the result of the leader election to our tool. Different SUTs have different requirements that must be met before they can accept writes. For example, some systems require a leader to be elected before they can ingest writes. These requirements must also be encoded in the CEM so that writes specified in the abstract schedule are correctly applied to the concrete SUT.

For each type of transition specified by the AEM, the CEM may have several ways of realizing that transition, allowing multiple concrete test sequences to be generated from a single abstract schedule. For example, some SUTs treat different types of failures differently (i.e., replica crash vs a network partition). The CEM may run the same schedule but select a different failure type at each $\mathcal{D} \rightarrow$. Similarly, there can be different options during $C \rightarrow$. Another place where CEM may have several options is what concrete set of writes, in terms of key names and values, will be used to realize the abstract writes in the AEM. In most cases, a set of unique values to a small set of keys suffices.

Below we show how our Q/C/Z-DRM's CEM interprets and injects events into a SUT for realizing divergence and convergence transitions to manifest our example bug. `setData(<k>, <v>)` sets <k> to <v>.

To realize, $\mathcal{D} \rightarrow [0, 1, 0]$:

- ① Crash A // no need to crash C, as it is already down
- ② `setData(<k3>, <v8>)` to B
- ③ Wait for commit on B
- ④ Crash B

To realize, $C \rightarrow [0, 2]$:

- ① Restart A
- ② Restart C
- ③ Wait for resync completion

3.3 DRM Examples

Overview. Table 3 shows the description of DRMs we have implemented to test ZooKeeper, MongoDB, and Redis in our experiments. We name the DRMs according to the following format: The first letter indicates whether the SUT requires a quorum of replicas (i.e., more than one-half of the replicas must be online) to receive write requests or not. Our DRM

models support both systems that require quorum (Q) and those that are stand-alone (S). The second is how replica failures are injected in the model. For failure methods, Modulo can forcibly kill replicas with the signal `SIGKILL` to simulate crash failures (C), suspend replicas with the signal `SIGSTOP` to simulate systems stalled due to a sudden burst of heavy load (S), prevent replicas from communicating to simulate link failures (L) or decommission replicas from a cluster to simulate replica replacement (D). The third is which SUT it is written for, either ZooKeeper (Z), MongoDB (M), or Redis (R). Models are named using the scheme `<quorum_requirement>/<failure_modes>/<SUT>`.

Also, the user-specified portion of each DRM is presented in Table 3. We had to manually write between 72 to 782 lines of code, which is the result of the effort trying to reduce the manual effort required for each DRM. We put the majority of the code overlapped across DRMs into library or template classes that users can use or extend. We believe we can further reduce the number of codes to write manually. Even for S/CL/R-DRM which required the largest codes for us to write has a large portion of the code that can be further implemented as a library or a template class. Therefore, we think the manual effort required to use DRMs is not heavy and it can become even lighter as the library and templates get mature.

Methodology. In implementing DRMs, we learned a couple of key lessons. First, one should write DRMs in a top-down approach. Think about the most general behavior first. Then, one can extend it by inheriting the most part while overriding only for differences. By doing so, users can reduce the amount of the code they need to write significantly. For instance, write a DRM that injects only one type of failure first. Then, users can write a DRM that can inject multiple types of failures by reusing many lines of code.

Second, focus on the behavior that matters to find target bugs. Users may have a specific type of CFBs they want to find foremost. For instance, ZooKeeper has suffered from errors in transaction log handling. To create more complicated cases, crashing and restarting is important because transaction log is backed by files where a new file is created everytime a new epoch begins. However, focusing on crash failures may not be effective for other systems that rely heavily on full resync using a single transaction log file or that may always use a snapshot resync. For those systems, exploring various network partition failures may be more effective.

Third, pay attention to configuration parameters. Distributed systems rely on various configuration parameter values and their behaviors depend on them. For example, Redis maintains a log of transactions failed replicas missed for a certain timeout period in case a failed replica comes back. Normally, this speeds up resync if the replica returns before the timeout. Modelling divergence behavior to wait longer than the timeout before triggering resync will cause the log to be prematurely discarded and replication fail silently without

attempting to use full resync or SNAP resync.

We now describe some interesting aspects of some models we built in more detail below.

Q/C/Z-DRM. This model is used for ZooKeeper, which requires a quorum of replicas to be online to start servicing requests. This DRM only models crash failures. Since a quorum is required, a $\mathcal{D} \rightarrow$ is only allowed when enough replicas to form a quorum are online, but at the same time $\mathcal{C} \rightarrow$ are not enabled when all replicas are online because there is no divergence to resolve. This model simply uses `kill -9` to send a `SIGKILL` signal, simulating crash failures. Since a quorum is required, the model is restricted in that it must ensure a quorum of replicas is available before it can start injecting write operations, otherwise the write operation will be rejected by the SUT. Also, after $\mathcal{C} \rightarrow$, it needs to pause before executing the next $\mathcal{D} \rightarrow$ in order to ensure that resync is complete. In ZooKeeper prior to version 3.5, log messages are scanned to confirm the existence of a leader that guarantees the resync completion. As of 3.5, it is not the case, so we use timeout.

Q/C/M-DRM. We reuse the same AEM as the Q/C/Z-DRM's. This demonstrates how AEMs can be reused for different SUTs. To confirm the resync completion, we use an MongoDB API call to query the internal document, "replSetGetStatus," to retrieve the timestamp of the latest transaction committed on each replica. Then, we wait until those timestamps of every replica becomes same. In case replicas never converge, potentially due to a CFB, it uses an internal time out.

S/S/R-DRM. This DRM does not require a quorum to start servicing clients. It models a chain replication where any replica can become a *sync source* and a *sync target*². Any write committed by a sync source will be replicated to its sync targets. Sync targets cannot have more than one sync source, but sync source can have multiple sync targets. This DRM considers suspend failures instead of crash failures. Also, this DRM starts with an initial state where no replica is connected with another replica. For recovery, we check if the replica is connected with another one and, if not, then we establish new links with other replicas. This model uses `kill -STOP` and `kill -CONT` to suspend and resume Redis processes, respectively. To confirm the completion of resync, this model employs the hybrid of 2 methods. First, it uses the `info` API call to read the `master_link_status`, `master_sync_in_progress`, `aof_rewrite_in_progress` and `rgb_bgsave_in_progress` variables that indicate whether resync is complete. Second, we found that just relying on these variables can cause the CEM to miss some cases when resync is complete. Thus, the CEM also uses a timeout to ensure forward progress. We believe this reduces unnecessarily long timeout, and demonstrates how flexible Modulo

²See: <https://redislabs.com/ebook/part-2-core-concepts/chapter-4-keeping-data-safe-and-ensuring-performance/4-2-replication/4-2-3-masterslave-chains/>

can be. Unlike ZooKeeper and MongoDB where resync is automatically triggered by recovering failures, Redis does not automatically trigger resync after a new replica joins. Instead, Redis requires a `slaveof` API call to be explicitly invoked to trigger resync. Thus, this CEM explicitly has replicas establish links between sync sources and a sync targets using the API call, as specified in schedules by its AEM.

S/L/R-DRM. This DRM models link failures only, causing a replica stop replicating data from its sync source. When it recovers a partitioned replica, it considers all possible scenarios of re-establishing replication links. This model simulates link failures using the Redis API command `slaveof no one`, which tells a replica that it has no sync source, causing it to stop replicating data from its sync source. When a link is re-established, the `slaveof` API is used.

S/CL/R-DRM. This model uses both crash and link failures. In addition, $\mathcal{C} \rightarrow$ can pick one of 2 kinds of resync strategies. One is *online resync* and the other one is *offline resync*. Online resync is a built-in resync mechanisms of Redis that gets triggered by re-establishing links between sync sources and sync targets. Offline resync, on the other hand, is a manual resync procedure that can be performed by an administrator. An administrator may manually copy the snapshot of a sync source to a sync target and then may have the sync target start off on the snapshot copy. Because, it considers both types of failures, it generates a larger state-space than the previous AEMs.

3.4 Implementation

Modulo is implemented in Java and comprises roughly 8.4K lines of code. Schedule generation is implemented in about 281 lines of code, and concrete execution takes about 766 lines of code. Our DRMs total 7.3K lines of code where the AEMs and CEMs consist of 2.8K and 4.6K lines of code, respectively, including DRM examples, a library and templates.

A significant part of the DRM implementation consists of SUT log parsing, API interaction and analysis code to infer the state of SUT replicas. To give a concrete example, ZooKeeper records whether a replica is a leader or not in its log message file as follows:

```
LEADING - LEADER ELECTION TOOK - 230
Follower sid: 0 : info : ...
Synchronizing with Follower sid: 0 ...
```

Similarly, a follower replica will log the following messages:

```
FOLLOWING - LEADER ELECTION TOOK - 217
Resolved hostname: 127.0.0.1 to address: ...
Getting a diff from the leader 0x100000009
```

Thus, the DRM can scan log message files of ZooKeeper looking for messages containing either LEADING or FOLLOWING

Name (Parameters)	AEM	CEM	Lines of Code (AEM/CEM/Total)
Q/C/Z-DRM ($numOps = \{1..5\}$, $numReplicas = \{3..5\}$)	Only considers crash failures. $C \rightarrow$ ensures the quorum exists before $\mathcal{D} \rightarrow$. Crashes all replicas at the end of $\mathcal{D} \rightarrow$	Using kill -9 to send SIGKILL for crash failures. Confirm the quorum exists before writes. Using log scanning to confirm the leader for versions before 3.5, but, as of 3.5, relying on timeout.	USER 54/59/113 LIB 339/620/959
Q/C/M-DRM (same as above)	Same as Q/C/Z-DRM	Using an API to retrieve “replSetGetStatus” and compare timestamps of the last transaction on each replica to wait for resync completion	USER 54/117/171 LIB 339/907/1246
S/S/R-DRM ($numOps = \{1..2\}$, $numReplicas = \{4\}$)	Only considers suspend failures. Considers all replicas initially partitioned. As recovering suspend failures, establish links between the recovered replica and an online replica.	Using kill -STOP and kill -CONT to simulate suspend and resume. Using the ‘info’ API and timeout to wait for resync completion. Using the ‘slaveof’ API to trigger resync	USER 33/39/72 LIB 955/1240/2195
S/L/R-DRM ($numOps = \{1\}$, $numReplicas = \{3\}$)	Only considers link failures between an arbitrary pair of replicas. Considers replicas initially connected in a single ‘slave chain.’	The ‘slaveof’ API is used for link failures and recoveries. Initially, forming links as a single slave chain.	USER 0/110/110 LIB 955/1240/2195
S/CL/R-DRM ($numOps = \{1, 2\}$, $numReplicas = \{2\}$)	Considers both link and crash failures. Considers two types of resync strategies: online resync and offline resync.	For the offline resync strategy, a script copying over snapshots and starting up a replica with the snapshot is used.	USER 405/377/782 LIB 955/1240/2195

Table 3: The summary/comparison of DRM Examples. The naming convention indicates the quorum requirement, failure modes, and the target SUT. Below the name, we also present modulo parameter values we used. AEM and CEM give more detailed descriptions of each subcomponent of the given DRM example. Only the differences of each DRM compared to the one above is specified. The rightmost column shows the lines of code (LOC) for user-specified portion of DRMs (USER) and for a library and templates users use (LIB)—only the part directly interfacing with user-specified portion is counted. Since each DRM may use different template, there can be difference in LOC for LIB.

indicating which role they have switched to. The DRM needs to keep track which replica is the leader in order to successfully inject write operations to cause divergence, because only the leader replica can ingest write operations.

4 Evaluation

We present our results from running Modulo on 3 mature, open-source, distributed storage systems: ZooKeeper, MongoDB, and Redis.

4.1 Bug Discovery

In Table 4, we summarize the CFBs found and tabulate the time Modulo took to find each of the bugs, as well as the number of transitions of the schedule manifesting the bug. More detailed description is provided in Appendix A. Those bugs labelled with “New!” had not been reported until we discovered. Our evaluation study has been conducted between 2017 and 2020—ZooKeeper Bug #1 and ZooKeeper Bug #2 were found in 2017, while ZooKeeper Bug #3, ZooKeeper Bug #4, and ZooKeeper Bug #5 were discovered in 2020. We note that some bugs are quite complex, requiring a specific sequence of more than 10 transitions to trigger the bug. Also, DRM state space is evaluated in terms of the number of schedules generated for different DRMs and different modulo parameter

values. Lastly, we show the state coverage measurement over time for the setting used to test the most recent ZooKeeper version.

4.2 Testing Performance

Table 4 also shows how much time our prototype took to find each bug mentioned above. The performance was measured on a machine with an 2.83GHz Intel Core2 Quad CPU and 8GB of RAM. We found that the limiting factor for the testing speed of Modulo is (1) the speed of the underlying distributed system and (2) how easily Modulo can infer that the system has converged after a $C \rightarrow$ so that it can inject a $\mathcal{D} \rightarrow$. In general, distributed systems do not prioritize speed during replication operations, and the lack of interfaces to infer when they are done can lead to slower testing as in the case of MongoDB. Without an interface that allows the CEM to tell if convergence had been achieved, Modulo had to check the timestamp of the keys on every replica to see if they were the same. In addition, MongoDB can take a long time to achieve convergence, forcing the CEM to use a very high timeout value (10 minutes in our experiments). In contrast, the CEMs for ZooKeeper and Redis can infer that convergence has occurred in other ways that do not require either checking or a timeout. We believe it should be possible to modify MongoDB to allow Modulo to infer whether convergence has

Bug ID	DRM	Root Cause	Elapsed Time	Time/Schedule	# of Trans.
ZooKeeper Bug #1(New!) [8]	Q/C/Z-DRM	Fail to truncate operations due to missing invocation	11 hours	33 sec	11
ZooKeeper Bug #2(New!) [9]	Q/C/Z-DRM	Fail to truncate operations due to file handling mistake	2 hours	39 sec	11
ZooKeeper Bug #3(New!) [10]	Q/C/Z-DRM	Fail to replicate operations due to an incomplete log	23 min	33 sec	7
ZooKeeper Bug #4(New!) [11]	Q/C/Z-DRM	Fail to truncate operations due to a pointer handling mistake	47 min	30 sec	10
ZooKeeper Bug #5(New!) [12]	Q/C/Z-DRM	Fail to truncate operations due to missing invocation	20 hours	37 sec	10
MongoDB Bug #1	Q/C/M-DRM	Fail to truncate operations due to incomplete timestamp information	18 min	6 min	3
MongoDB Bug #2(New!) [36]	Q/C/M-DRM	Fail to replicate operations due to incomplete protocol design	4 hours	5 min	5
Redis Bug #1 [25]	S/S/R-DRM	Fail to invoke snapshot sync due to incomplete protocol design	6 hours	6 min	6
Redis Bug #2 [53]	S/CL/R-DRM	Fail to replicate operations due to lacking resync related information	11 min	14 sec	4
Redis Bug #3 [53]	S/CL/R-DRM	Fail to replicate operations due to lacking resync related information	2 min	6 sec	3
Redis Bug #4 [22]	S/L/R-DRM	Fail to truncate operations due to incomplete protocol design	2 min	33 sec	2

Table 4: CFB Analysis Summary.

DRM	numOps	numReplicas	# of Schedules
Q/C/Z	1	3	6
	2	3	80
	3	3	1035
	4	3	13381
	5	3	172993
	3	4	3428
	3	5	54655
S/S/R	2	4	13586
S/L/R	2	3	263
S/CL/R	1	2	8
	2	2	96

Table 5: DRM State Space Size (# of Schedules).

occurred without the needing to rely on timeouts and intend to explore this in the future.

4.3 DRM State Exploration

Table 5 shows several examples of the DRM state space size in terms of the number of schedules generated. As the number of *numOps* or *numReplicas* increase, the number of schedules generated quickly grows. The largest number of schedules were generated for Q/C/Z-DRM with *numOps* = 5 and *numReplicas* = 3 which we used to test ZooKeeper 3.4.11 and found the example bug. Nevertheless, as we reason in §5, the state space for Modulo to search is much smaller than explicit state model-checkers.

In Figure 4, we measured how state space coverage is increased during our evaluation finding bugs in the version of ZooKeeper, 3.7. We fixed *numReplicas* at 3 and varied *numOps* from 1 to 4, which increases the time taken. For each *numOps* setting, we ran tests separately one after another. Bugs can be found by schedules with no particular probability distribution. Considering this, it is important to run each test exhaustively not to miss any bug. Our design choice to split AEM and schedule generation from the con-

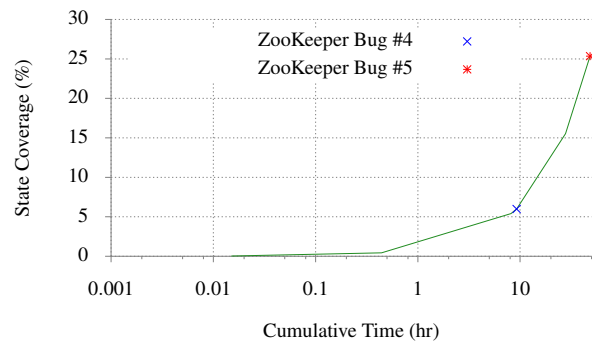


Figure 4: DRM State Coverage graph. We show the state coverage measurements from testing ZooKeeper 3.7. X-axis shows the cumulative time and Y-axis shows the state coverage ratio based on the number of schedules executed. Also, we mark when we found ZooKeeper Bug #4 and ZooKeeper Bug #5.

crete test execution by CEM also enables easy parallelization. Indeed, we when parallelize our tests, we get a linear increase in throughput.

5 Discussion

Modulo is designed to detect CFBs efficiently by only exploring system states that are relevant to its DRMs, which only model states and events relevant to $C \rightarrow$ and $D \rightarrow$ state transitions. In contrast to traditional system model-checking [27,29,35,41,42,44,52,61], Modulo is less complete, meaning it may miss bugs that manifest due to events like lock acquisition and thread interruption, as these are not captured by $C \rightarrow$ and $D \rightarrow$ in DRMs. However, by abstracting away states and events not relevant to DRMs, Modulo is able to explore considerably deeper bugs that, in some cases, take 10 or more transitions to manifest. Furthermore, Modulo finds

these bugs in real, concrete systems rather than in abstract models, making reproduction and confirmation of bugs much simpler.

Compared to distributed systems random testing, such as that employed by Jepsen [38], Modulo is more systematic, complete and exhaustive. Random testing can find many crucial bugs by randomly injecting external events; however, it is neither systematic, complete nor exhaustive for the following reasons. First, it does not model how the underlying distributed system works, including key concepts such as divergence and convergence. Hence, random testing may miss corner cases regarding divergence and convergence. Second, fuzzing does not control nondeterministic events. For instance, without delaying the timing of a crash-failure injection, a replica may crash before ensuring a write injected before the crash is committed. Third, random testing cannot reproduce bugs the way Modulo can, making the analysis and reproduction of bugs more challenging.

Modulo significantly reduces the severity of state explosion by taking a simple abstract model and mapping it onto transitions on a real concrete system. This allows Modulo to find deeper bugs than systems that attempt to explore more complex state spaces. To illustrate, Modulo's AEM's typically model 3-replica systems with between 1 and 5 writes. Before each write in the sequence, a replica can typically (1) do nothing, (2) receive or replicate the write operation, or (3) failure/recovery (e.g. crash/restart). Since replicas may have an arbitrary number of transitions where they may do nothing, we typically cap the maximum number of transitions in a replica an AEM may have at around 8. Thus we can estimate a rough upper bound for the state space of such a system as $3^{(8 \times 3)}$ or roughly 300 million. In practice, the number of reachable states is far less, because replicas can accept writes only when they are online, and other AEM-specific restrictions such as quorum requirements further limit the transitions that replicas may execute. In practice, Modulo's targeted approach leads to have AEMs produce anywhere from few schedules to tens of thousands of schedules, which is about 10 to 10^5 of states. In comparison, models checked by explicit state model checkers may contain more than 10^{20} or even 10^{250} states [17]. Our relatively small DRMs mean that Modulo is able to explore more scenarios than traditional model checkers could in given time, increasing the likelihood of finding target bugs, without exploring states that are irrelevant to find those specific types of bugs.

Meanwhile, we also note that a targeted approach can be seen as a disadvantage for finding many of various types of bugs. Indeed, as Modulo is targeted to find a specific type of CFBs, the number of bugs we found from our evaluation is relatively low. Nevertheless, we envision that the number of bugs found by Modulo can be increased, as developers who are more expert of each SUT can develop a larger collection of DRMs that are more effective to explore corner cases.

We acknowledge that Modulo does depend on domain

knowledge of the system-under-test to specify DRMs, and this would also be required to apply Modulo to other types of distributed systems. In our experience, a single user without previous experience may conservatively take about 2 weeks to learn about the system-under-test and 2 weeks to write the first DRM. When we first applied Modulo to Redis, using the same model as ZooKeeper and MongoDB did not lead to bugs, because Redis, started as an in-memory key-value store, has a simpler persistent storage mechanism. Subsequently, we found exploring suspend or link failures enabled Modulo to trigger more complex functionality. Modulo's methodology, abstraction and concrete execution, is not necessarily restricted to key-value stores but can be applied to other distributed systems. Paxos-based systems can be effectively tested using Modulo by modelling message/thread interleaving alongside failure injections.

Our main limitation is that the schedules generated by the abstract model must be run on the real SUT, which executes much slower than an abstract model would. In addition, many operations require pauses and timeouts before they can complete. For example, a system may not consider a replica failed until a certain time has passed, and leader election must complete after replicas recover before writes can be ingested by the system. Since they are not always possible to avoid, in practice we find that these timeouts are the ultimate limit on how fast Modulo can find bugs. We think virtualizing clocks and fast-forwarding time may help [42].

6 Related Work

Distributed Systems Testing. Some previous proposals for distributed systems testing employ state-space exploration. However, their state-spaces are more focused on interleaving concurrent internal events, such as thread scheduling or network message delivery [20, 28]. Also, previous work presents a tool for injecting network-partitioning failures for cloud systems [7], yet it does not inject crash failures, and it requires an OpenFlow-capable hardware component to simulate network-partitioning. Jepsen is an open-source testing tool that injects various types of failures into distributed database systems [38]. However, unlike Modulo, Jepsen randomly generates inputs, which implies that the state space for input sequences cannot be efficiently reduced without missing corner cases and, in some cases, reproducing bugs may be very difficult due to the nondeterministic ordering of events that Jepsen does not control. Furthermore, Jepsen does not record any information about the state exploration, therefore it cannot provide any guarantee about the state-space coverage. There has been a work that focuses on interleaving low-level file system operations across distributed replicas along with crash failures [6], but it is limited to crash injection only and does not test the divergence and convergence behaviors of distributed systems.

Model-based testing systematically derives test cases from

an abstract model of the SUT [14, 37, 45, 48, 49, 54]. Dalal et al. [18] devised a method that can generate various input parameter values for tests from the abstract model called the Test Data Model. A technique that can derive test cases for system testing from an UML statechart was developed by Offutt et al. [47]. Also, Gargantini et al. [24] propose to use model checking to derive test cases from an abstract model. In addition, Andrews et al. [13] came up with a technique that models a web application as a finite state machine to generate tests. Also, Yang et al. [63] applied model-based testing to find security flaws in about 500 implementations of OAuth 2.0. More recently, Davis et al. [19] used model-based testing to ensure specification-implementation conformance of distributed systems. However, previous proposals for model-based testing do not look for CFBs. Also, none of the existing model-based studies test the divergence and convergence of replicated distributed storage systems as the SUT.

Distributed Systems Model Checking. Model checking has been extensively studied and used to prove the correctness of complex systems and to find bugs in them. Clarke and Emerson [16] were the first to propose model checking, which exhaustively explores the state space of abstract models specified in temporal logic. Dill developed a model checker called Murphi [21], which is used to prove the correctness of various systems, including distributed shared memory systems. SPIN is another popular abstract model-checking tool [30]. More recently, Lamport [40] developed TLA+, a specification language, and TLC, a model checker for TLA+, which has been used by Amazon, showing the practicality of model checking in the industry. Nevertheless, abstract model-checking cannot find bugs in implementations directly.

Concrete model-checking is used to employ an implementation as a model to explore directly. Musuvathi et al. [44] proposed a concrete model-checker using implementations as the model to verify. Godefroid [26] also proposed the same idea around the same time. Killian et al. [35] devised a technique that enables checking for not only safety properties but also liveness properties. Lin et al. [42] developed a black-box concrete model-checker that does not need to know about the source code by interposing events at the interface layer between the target system and the underlying operating system. Model checking not only proves the correctness of the system but also predicts if the implementation execution is driving the system to faulty states, steering the system execution away to prevent this [61]. Simsa et al. [52] explored the generalization of concrete model-checking to provide the flexibility for determining the level of controls for nondeterminism. Recently, concrete model-checkers have been improved to detect deep bugs by exploring scenarios involving multiple failures [41, 43]. Unlike Modulo, previous concrete model-checkers explore various tightly controlled sequences of concurrent internal events.

7 Conclusion

Modulo mitigates the traditional state-explosion problems of systematic model-checking approaches to find CFBs by abstracting away all states and state transitions that are not related to the concepts of convergence and divergence. Modulo applies schedules derived from state explorations of small abstract models of such systems to real distributed systems. Our work identified several factors that lead to such bugs, which include (1) employing several resync or failure-handling mechanisms whose interactions are difficult to foresee, (2) hard limits or inadequate designs for handling large amounts of divergence, and (3) assumptions about length of time that replicas may have failed and failures that span events like leader transitions. While it is beneficial to generate counterexamples that trigger the bugs on real systems, we find that this also slows down the speed at which Modulo can find bugs, as it must examine the state of the system at to determine if it can inject the next input or not, and it must wait for the distributed system itself to complete its replication operations before adding divergence.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Nathan Bronson, for their wonderful guidance and feedback. We also thank Ashvin Goel and Eyal de Lara for their helpful comments. This research was supported by NSERC Discovery Grant RGPIN-2018-05931, a Canada Research Chair, and NSF CNS-1749711.

References

- [1] Apache HBase. <https://hbase.apache.org>.
- [2] Couchbase Server. <https://www.couchbase.com>.
- [3] MemcachedDB. <http://memcachedb.org>.
- [4] MongoDB. <https://www.mongodb.com/>.
- [5] Riak. <http://basho.com/products>.
- [6] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayanan Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated crash vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16*, pages 151–167, Berkeley, CA, USA, 2016. USENIX Association.
- [7] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of*

the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'18, pages 51–68, Berkeley, CA, USA, 2018. USENIX Association.

- [8] anaud. Synchronization code on the follower does not properly truncate uncommitted write resulting in data inconsistency, 2017. <https://issues.apache.org/jira/browse/ZOOKEEPER-2945>.
- [9] anaud. The truncate() function in filetxnlog.java may fail to properly remove an uncommitted write resulting in data inconsistency, 2017. <https://issues.apache.org/jira/browse/ZOOKEEPER-2946>.
- [10] anaud. Convergence fail when a follower tries to resync with a leader having incomplete commitlog, 2020. <https://issues.apache.org/jira/browse/ZOOKEEPER-3972>.
- [11] anaud. Convergence fails when a follower missed the committedlog synching with the leader if it was an old leader and the leader falls back to send snapshot., 2020. <https://issues.apache.org/jira/browse/ZOOKEEPER-3946>.
- [12] anaud. truncate in filetxnlog.java is buggy and fails to correctly truncate a file containing a single transaction only the follower saw, 2020. <https://issues.apache.org/jira/browse/ZOOKEEPER-3947>.
- [13] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with FSMs. *Softw. Syst. Model.*, 4(3):326–345, July 2005.
- [14] Lionel C. Briand and Yvan Labiche. A UML-based approach to system testing. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, UML '01, pages 194–208, London, UK, UK, 2001. Springer-Verlag.
- [15] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, pages 147–158, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
- [16] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.
- [17] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, page 176–194, Berlin, Heidelberg, 2001. Springer-Verlag.
- [18] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM.
- [19] A. Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. Extreme modelling in practice. *Proceedings of the VLDB Endowment*, 13(9):1346–1358, May 2020.
- [20] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering bugs in distributed storage systems during testing (not in production!). In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 249–262, Berkeley, CA, USA, 2016. USENIX Association.
- [21] David L. Dill. The Murphi verification system. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 390–393, London, UK, UK, 1996. Springer-Verlag.
- [22] fdingiit. Questions about potential data inconsistency / unexpected FSYNC base on 4.0.2, 2017. <https://github.com/antirez/redis/issues/4407>.
- [23] Camille Fournier. leader/follower coherence issue when follower is receiving a diff, 2010. <https://issues.apache.org/jira/browse/ZOOKEEPER-962>.
- [24] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. *SIGSOFT Softw. Eng. Notes*, 24(6):146–162, October 1999.
- [25] GeorgeBJ. Replication inconsistent issue, 2015. <https://github.com/antirez/redis/issues/2694>.
- [26] Patrice Godefroid. Model checking for programming languages using Verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.
- [27] Rachid Guerraoui and Maysam Yabandeh. Model checking a networked system without the network. In *Proceedings of NSDI'11: 8th USENIX Symposium on Networked Systems Design and Implementation*, page 225, 2011.
- [28] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruva Borthakur. FATE and DESTINI: A framework for cloud

- recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 238–252, Berkeley, CA, USA, 2011. USENIX Association.
- [29] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 265–278. ACM, 2011.
- [30] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.
- [31] Hailin Hu. Inconsistent query results between primary and secondary, 2017. <https://jira.mongodb.org/browse/SERVER-31663>.
- [32] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.
- [33] Jacky007. Data inconsistency when follower is receiving a diff with a dirty snapshot, 2012. <https://issues.apache.org/jira/browse/ZOOKEEPER-1549>.
- [34] Vishal Kathuria. Data inconsistency when the node(s) with the highest zxid is not present at the time of leader election, 2011. <https://issues.apache.org/jira/browse/ZOOKEEPER-1154>.
- [35] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of NSDI'07: 4th USENIX Symposium on Networked Systems Design and Implementation*. NSDI, 2007.
- [36] Beom Heyn Kim. Initial sync not replicating old oplog entries may have a stale node give up resync and permanently stay in recovering state, 2018. <https://jira.mongodb.org/browse/SERVER-35774>.
- [37] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha. Test cases generation from uml state diagrams. *IEE Proceedings - Software*, 146(4):187–192, Aug 1999.
- [38] Kyle Kingsbury. Distributed systems safety research. <https://jepson.io/>.
- [39] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.
- [40] Leslie Lamport. Specifying concurrent systems with TLA+, 1999.
- [41] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 399–414, 2014.
- [42] Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: transparent model checking of unmodified distributed systems. In *Proceedings of NSDI'09: 6th USENIX Symposium on Networked Systems Design and Implementation*. NSDI, 2009.
- [43] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. FlyMC: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 20:1–20:16, New York, NY, USA, 2019. ACM.
- [44] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review*, 36(SI):75–88, 2002.
- [45] Clementine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jezequel. Automatic test generation: A use case driven approach. *IEEE Trans. Softw. Eng.*, 32(3):140–155, March 2006.
- [46] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Commun. ACM*, 58(4):66–73, March 2015.
- [47] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of the 2nd International Conference on The Unified Modeling Language: Beyond the Standard*, UML'99, pages 416–429, Berlin, Heidelberg, 1999. Springer-Verlag.
- [48] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Softw. Test., Verif. Reliab.*, 13(1):25–53, 2003.
- [49] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 105–118, New York, NY, USA, 1992. ACM.
- [50] Salvatore Sanfilippo. Redis. <https://redis.io/>.

- [51] Andy Schwerin. Multi-updates may fail to detect replica set primary step-down, leading to inconsistency., 2014. <https://jira.mongodb.org/browse/SERVER-12516>.
- [52] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic evaluation of distributed systems. In *Proceedings of the 5th International Conference on Systems Software Verification, SSV'10*, page 3, USA, 2010. USENIX Association.
- [53] soloestoy. Redis 4.x PSYNC2 & RDB: data inconsistency between master and slave, bug located, 2017. <https://github.com/antirez/redis/issues/4316>.
- [54] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Trans. Softw. Eng.*, 22(11):777–793, November 1996.
- [55] Jeremy Stribling. Missing data after restarting+expanding a cluster, 2011. <https://issues.apache.org/jira/browse/ZOOKEEPER-1319>.
- [56] Jeremy Stribling. Data inconsistencies and unexpired ephemeral nodes after cluster restart, 2012. <https://issues.apache.org/jira/browse/ZOOKEEPER-1367>.
- [57] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project Voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies, FAST'12*, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.
- [58] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-Based Service Level Agreements for Cloud Storage. In *The 24rd ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.
- [59] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.
- [60] Ryan Witt. Secondary keeps getting into an inconsistent state, 2014. <https://jira.mongodb.org/browse/SERVER-13222>.
- [61] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI*, volume 9, pages 229–244, 2009.
- [62] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14*, pages 157–168, New York, NY, USA, 2014. ACM.
- [63] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu. Model-based security testing: An empirical study on OAuth 2.0 implementations. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, ASIA CCS '16*, pages 651–662, New York, NY, USA, 2016. ACM.
- [64] Christian Ziech. Data loss after truncate on transaction log, 2012. <https://issues.apache.org/jira/browse/ZOOKEEPER-1489>.

A Bug Description

A.1 ZooKeeper Bugs

We discovered 5 new bugs in ZooKeeper by finding 2 new bugs in version 3.4.11, 1 new bug in version 3.5.8 and 2 new bugs in version 3.7.0. We reported them to the ZooKeeper developers and reported bugs were designated as ZooKeeper Bug #1, ZooKeeper Bug #2, ZooKeeper Bug #3, ZooKeeper Bug #4 and ZooKeeper Bug #5 [8–12]. We got confirmation for ZooKeeper Bug #1 and ZooKeeper Bug #3. ZooKeeper Bug #1 is described in §2.3. All our experiments used 3 replicas. Initially, all 3 replicas are online and synchronized with a initial key-value set. Also, the Q/C/Z-DRM implementation for ZooKeeper crashes all replicas after each $\mathcal{D} \rightarrow$ and implements $C \rightarrow$ by restarting replicas in the quorum, which automatically triggers resync between each replica and the elected leader.

As described earlier, ZooKeeper implements 2 different resync mechanisms: DIFF resync and SNAP resync. Conflict-resolution logic often fails to correctly truncate transaction logs when SNAP resync is used, which results in persistent inconsistency. This problem is further exacerbated by an apparent reluctance on the part of the developers to truncate transaction logs, perhaps out of a conservative preference not to lose data unless absolutely necessary, which results in cases where logs should have been truncated but were not. Other complexities, such as using multiple log files instead of a single log file and incorrect assumptions about those files, contributed to other bugs. We saw similar bugs are recurring and stem from the similar portion of the resync implementation. It shows that the complexity of the resync mechanisms in ZooKeeper has been the major source of CFBs.

A.2 MongoDB Bugs

We run Modulo on MongoDB version 3.0.0 and discovered 2 bugs. One was fixed in later versions of MongoDB by upgrading their replication protocol, Replica Set Protocol (version), from Protocol Version 0 to Protocol Version 1. The

other one was new and we reported it on the MongoDB Bug Database [36]. We used 3 replicas in the DRM and the same initial state as we did with ZooKeeper.

The first MongoDB bug we discussed occurred because the developers failed to anticipate the situation where a primary commits some operations that have not been replicated to other replicas, operations only the primary is thus aware of. Modulo found the bug and the bug had never previously reported. However, the bug does not manifest on the latest version of MongoDB. On further inspection, we found that Replica Set Protocol Version 0, which had this bug, was replaced by Replica Set Protocol Version 1, which was implemented in MongoDB version 3.2 and became the default protocol after version 3.6. Thus, this particular bug was not fixed directly by developers, but instead eliminated when the afflicted protocol was replaced by a completely new protocol. With Modulo's bug report, we estimate that the bug could have been fixed by changing 10's of lines of code, but instead was fixed when the entire protocol was re-implemented, which required changing 12 files containing roughly 7.4K lines of code.

The second bug demonstrates the perils of simultaneously using several resync mechanisms. Having several resync mechanisms allows MongoDB to select among them to improve efficiency, but the mechanisms have slightly different side effects, which, under the right circumstances, can combine to put the system into an unrecoverable state.

A.3 Redis Bugs

Modulo found 4 CFBs [22, 25, 53] in Redis versions 2.8.0 and 4.0.0. Upon further examination, we found all had been previously reported.

In Redis, crash failures always lead to the SNAP resync, which simply replicates the entire state of the sync source to the sync target. As mentioned earlier, this trivially guarantees convergence since the sync target is now a mirror of the sync source. To trigger more complex resync mechanisms, Modulo required DRMs that could exercise other replication and failure recovery mechanisms that Redis provides. Unlike Zookeeper and MongoDB, this required the development of 3 different DRMs, and each DRM was responsible for finding at least 1 CFB. We found it useful to have separate DRMs as each DRM could separately exercise some of Redis' features, while combining them would have resulted in a larger state space and more schedules to explore.

In terms of time and effort, Redis took the most: a novice Redis user took a couple of weeks to initially write each DRM, where most of the time was spent understanding Redis' mechanisms and API. The Redis DRMs are also roughly 2-3 times larger and more complex than ZooKeeper and MongoDB DRMs because replicas may specify any sync source to resync from, giving more possibilities. Qualitatively, we feel the effort to construct DRMs is similar to that of writing a

specification in a formal specification language such as TLA+, which has been cited as an acceptable cost by developers at Amazon [46]. The user needs to understand the important properties of the system they want to test, and be able to abstract them away from implementation details. In addition, the user must also be able to reduce a system to a smaller number of replicas and smaller number of keys to reduce the state space. However, unlike model checkers such as TLA+, which run on an abstract state machine representation of the SUT, Modulo marries the advantages of a reduced state space produced by the manual abstraction, with the ability to reproduce the bugs found using a counter-example of real inputs that can be run on the concrete system.

SoftTRR: Protect Page Tables against Rowhammer Attacks using Software-only Target Row Refresh

Zhi Zhang¹, Yueqiang Cheng², Minghua Wang³, Wei He^{4,7}, Wenhao Wang^{4,7}✉, Surya Nepal¹, Yansong Gao⁵, Kang Li³, Zhe Wang^{6,7}, and Chenggang Wu^{6,7}

¹CSIRO's Data61, Australia

²NIO Security Research

³Baidu Security

⁴State Key Laboratory of Information Security, Institute of Information Engineering, CAS

⁵Nanjing University of Science and Technology, China

⁶State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS

⁷University of Chinese Academy of Sciences

Abstract

Rowhammer attacks that corrupt level-1 page tables to gain kernel privilege are the most detrimental to system security and hard to mitigate. However, recently proposed software-only mitigations are not effective against such kernel privilege escalation attacks.

In this paper, we propose an effective and practical software-only defense, called SoftTRR, to protect page tables from all existing rowhammer attacks on x86. The key idea of SoftTRR is to refresh the rows occupied by page tables when a suspicious rowhammer activity is detected. SoftTRR is motivated by DRAM-chip-based target row refresh (ChipTRR) but eliminates its main security limitation (i.e., ChipTRR tracks a limited number of rows and thus can be bypassed by many-sided hammer [17]). Specifically, SoftTRR protects an unlimited number of page tables by tracking memory accesses to the rows that are in close proximity to page-table rows and refreshing the page-table rows once the tracked access count exceeds a pre-defined threshold. We implement a prototype of SoftTRR as a loadable kernel module, and evaluate its security effectiveness, performance overhead, and memory consumption. The experimental results show that SoftTRR protects page tables from real-world rowhammer attacks and incurs small performance overhead as well as memory cost.

1 Introduction

Rowhammer is a software-induced dynamic random-access memory (DRAM) vulnerability that frequently accessing (i.e., hammering) DRAM aggressor rows can induce bit flips in neighboring victim rows. An attacker can hammer aggressor rows to corrupt different types of sensitive objects on victim

rows without access to them, breaking memory management unit (MMU)-based memory protection, achieving privilege escalation [13,46,62] or leaking sensitive information [11,37]. Of the many sensitive objects that have been corrupted by the rowhammer attacks, page table corruption is the most detrimental to system security, making kernel privilege escalation attacks the mainstream [57]. To date, kernel privilege escalation attacks [13,22,46,53,59,62] focus on corrupting level-1 page table entry (L1PTE) and some of them have been demonstrated to gain kernel privilege from unprivileged applications [13,46,62], or even from JavaScript in webpages [22].

Multiple software-only mitigation schemes [12,34,57] can be used to mitigate the kernel privilege escalation attacks. Compared to hardware defenses [30,38,40,49], software-only schemes have the appeal of compatibility with existing hardware, allowing better deployability. However, existing software-only mitigations require modifications to memory allocator and they are not effective against all the kernel privilege escalation attacks. Specifically, CATT [12] and CTA [57] are vulnerable to a recent privilege escalation attack (PThammer [62]) that targets L1PTE. ZebRAM [34] assumes that bit flips occur in a victim row that is one-row from hammered aggressor row(s), making itself unable to defend against (kernel privilege escalation) rowhammer attacks where a victim row is no less than 2-row from the hammered rows [32,62]. To this end, we ask:

Is there an effective and practical software-only defense that protects page tables against rowhammer attacks?

Our Contributions. In this paper, we provide a positive answer to the question. We propose a new software-only defense that defends against all existing kernel privilege escalation attacks on x86, called SoftTRR. SoftTRR is motivated by

a hardware defense, i.e., ChipTRR (known as TRR in the DRAM standards [30, 40]). ChipTRR is designed to count rows' activations and refreshing adjacent rows to suppress bit flips if the activation counts reach a pre-defined threshold. ChipTRR was believed to eliminate the rowhammer effect in present-day DDR4-based systems, until it was completely circumvented by [17].

We observe that the root cause of failure of ChipTRR is that it tracks a limited number of rows. Thus, bit flips are still possible when multiple rows are being hammered and the number of hammered rows is larger than the tracked rows (i.e., *many-sided hammer* [17]). SoftTRR addresses this limitation by monitoring and tracking all rows neighboring (victim) rows containing page tables. SoftTRR leverages MMU-enforced virtual memory subsystem to frequently track memory accesses to any rows adjacent to page-table rows, and refreshes page-table rows when necessary, making SoftTRR effective in preventing rowhammer from breaking page table integrity.

Specifically, MMU is an essential component of modern processors that supports OS kernel to enforce memory isolation. With the assistance from MMU, the kernel, configures page tables, mediates every memory access from user space, and captures any unauthorized access that triggers a hardware exception. On top of that, the kernel can capture the memory access where relevant page tables have an unused `rsrv` bit set (see page fault handler in Section 4.3). With this observation, SoftTRR uses the kernel as the root of trust and frequently configures page tables with the `rsrv` bit set to track memory accesses to rows that neighbor rows of page tables. When the tracked memory-access counters reach a pre-determined limit, corresponding page-table rows will be refreshed. By SoftTRR's design, an adjacent or neighboring row can be multiple-row from a page-table row, thus voiding the above assumption of one-row-distance between victim and aggressor rows made by ZebRAM [34]. In our implementation, the adjacent rows are up to 6-row away from the aggressor rows, the largest row distance that has been observed so far [32].

Our prototype implementation of SoftTRR is a loadable kernel module (LKM) without any modification to the kernel. The LKM has about 1700 source lines of code and it has been deployed into three Linux systems where underlying hardware have either DDR3 or DDR4 modules. We evaluated SoftTRR-deployed systems in terms of security effectiveness, performance, memory consumption and robustness. The experimental results show that SoftTRR is effective in mitigating kernel privilege escalation attacks. Besides, SoftTRR incurs low overhead on the tested benchmarks and its memory consumption is within hundreds of KiB in a real-world use case of LAMP (i.e., Linux, Apache, Mysql and PHP). We also validate the robustness of a SoftTRR-enabled system using system-call stress tests, results of which show that the system runs as stable as a vanilla system.

In summary, the main contributions are as follows:

- We introduce SoftTRR to defend against rowhammer at-

tacks on page tables. Compared to prior works, SoftTRR is an effective and practical software-only mitigation scheme.

- We implement a lightweight SoftTRR prototype to collect page tables, track memory access, and refresh target page tables by leveraging MMU and OS kernel features.
- We evaluate SoftTRR's effectiveness against 3 representative rowhammer attacks, its performance overhead and memory consumption. The experimental results show that SoftTRR successfully protects page tables against the attacks, and incurs negligible overhead and memory cost.

2 Background and Related Work

In this section, we first describe DRAM and its address mapping. We then present the rowhammer vulnerability as well as its hardware and software defenses. Please refer to [42, 63] for rowhammer surveys.

2.1 DRAM

The main memory of most modern computers uses DRAM. Memory modules are usually produced in the form of dual inline memory module (DIMM), where both sides of the memory module have separate electrical contacts for memory chips. Each memory module is directly connected to the CPU's memory controller through one of the two channels. Logically, each memory module consists of two ranks, corresponding to its two sides, and each rank consists of multiple banks. A bank is structured as arrays of memory cells with rows and columns.

Every cell of a bank stores one bit of data whose value depends on whether the cell is electrically charged or not. As the charge stored in the cell disperses over time, every cell's charge must be restored or refreshed periodically in a specified time period (i.e., t_{REFW}), a typical value of which is 64 milliseconds (ms).

DRAM Address Mapping. The memory controller decides how physical-address bits are mapped to a DRAM address. A DRAM address refers to a 3-tuple of *bank*, *row*, *column* (DIMM, channel, and rank are included into the *bank* tuple field). As this mapping is not publicly documented on the Intel processor platform, it has been reverse-engineered by multiple works [44, 45, 55, 59].

2.2 Rowhammer Vulnerability

Kim et al. [33] are the first to perform a large scale study of rowhammer on DDR3 modules, results of which have shown that the vulnerability can be triggered by software accesses, that is, frequently accessing rows of $i + 1$ and $i - 1$ (i.e., aggressor rows) cause bit flips (i.e., charge leakage) in row i (i.e., victim row).

There are four hammer patterns in existing works. First, *double-sided hammer* refers to a case where two adjacent rows of the victim row are hammered simultaneously, which is the most effective hammer pattern in inducing bit flips on DDR3 modules [46]. Second, *single-sided hammer* randomly picks two aggressor rows in the same bank and hammers them [46]. Third, *one-location hammer* selects a single aggressor row for hammer. This hammer pattern only applies to certain systems where the DRAM controller employs an advanced policy (i.e., the closed-page policy) to optimize performance [21]. Last, *many-sided hammer* chooses more than two aggressor rows within the same bank for hammer. The aggressor rows are usually separated by one row and two out of them are adjacent to the victim row [17, 29].

2.3 Rowhammer Defenses

Hardware Solutions. Existing hardware solutions employed by the industry can be summarized into three main categories. The first is to decrease the DRAM refresh period [33] to refresh all DRAM rows more frequently. For instance, three computer manufacturers (HP [25], Lenovo [39] and Apple [3]) deployed firmware updates to decrease the refresh period from 64 ms to 32 ms. However, *clflush-free* rowhammer attacks [5] still induce bit flips in the reduced refresh period. Decreasing the refresh period by more than 7x can make the rowhammer impossible but it will impose unacceptable overhead to the systems [33]. The second one is proposed by Intel [28] that leverages Error Correcting Code (ECC) memory to correct single-bit errors and detect double-bit errors. However, ECC has been reverse engineered and is vulnerable to rowhammer [15]. The last is to track row's activation count and various approaches have been proposed [30, 33, 38, 40, 43, 47–49, 60]. Among them, ChipTRR [30, 40] was adopted by recent DDR4 manufacturers but it has been reverse-engineered and defeated [17, 23, 29]. None of other approaches are widely deployed due to their limitations (e.g., significant area cost or performance downsides) [7].

Software Defenses. Software defenses include both mitigation and detection techniques. As sensitive data is required to be within victim rows for exploitation, existing mitigation techniques modify memory allocator and enforce DRAM-aware memory isolation at different granularity [9, 12, 34, 52, 54, 57]. CATT [12] implements DRAM isolation between user and kernel memory. CTA [57] provides a dedicated DRAM region for level-1 page tables. ZebraRAM [34] isolates rows of sensitive data in a zebra pattern. These defenses can prevent page tables from being hammered. Albeit on different hardware, SoftTRR has an averaged overhead of 0.75% on SPECint 2006 (see Appendix A), similar to that of CATT [12] and CTA [57]. However, ZebraRAM has a much higher overhead of 4%–5%. ALIS [52] isolates DMA memory to prevent the remote rowhammer attack [52] targeting a memcached application. RIP-RH [9] provides DRAM

isolation for local user processes.

Anvil [5] utilizes CPU performance counters to monitor cache miss rate and detects a rowhammer attack, as typical rowhammer attacks incur frequent cache misses. However, Anvil is prone to false positives [12, 57]. Besides, its current implementation cannot detect the PThammer attack [62]. The other detection technique is RADAR [61]. As rowhammer attacks exhibit recognizable rowhammer-correlated sideband patterns in the spectrum of the DRAM clock signal, RADAR leverages peripheral customized devices to capture and analyze the electromagnetic signals emitted by a DRAM-based system.

3 SoftTRR: Software-only Target Row Refresh

We discuss threat model and assumptions in Section 3.1, design principles in Section 3.2 and design overview in Section 3.3. Section 4 describes implementation details.

3.1 Threat Model and Assumptions

Our primary goal is to protect page tables and guarantee that an adversary cannot corrupt them to gain kernel privilege through rowhammer on x86 architectures. In our implementation of SoftTRR, we focus on protecting level-1 page tables (LIPTs), the same goal as in CTA [57], because all existing page-table-oriented rowhammer attacks aim at corrupting LIPTs. Even when higher levels of PTs are corrupted, they are hard to be exploited (see details in CTA [57]). In spite of that, SoftTRR can be extended to protect other levels of page tables and we discuss it in Section 7.

We assume the kernel as the root of trust, and the kernel module implementing SoftTRR is well protected. We consider threats coming from both local adversaries and remote adversaries. A local adversary resides in a low privilege user process and thus can execute arbitrary code within her privilege boundary. A remote adversary stays outside by launching an attack, e.g., through a website with JavaScript.

The DRAM address mappings and in-DRAM address remappings can be reverse-engineered using prior works [14, 44, 55, 59] and they are assumed to be available. Besides, previous software-only rowhammer defenses [9, 12, 34, 57] consider that hammering row_i only affects row_{i+1} and row_{i-1} , which however is not consistent with a recent work by Kim et al. [32]. Particularly, they performed a comprehensive study of 1580 DRAM chips (300 DRAM modules in total) from three major DRAM manufacturers and found that bit flips can occur in rows that are up to 6-row away from the hammered row_i . SoftTRR by design protects rows of page tables from being flipped by rows that are N-row away and its current implementation allows that the distance between an adjacent row and an LIPT row ranges from 1-row to 6-row, the largest row distance observed by Kim et al. [32].

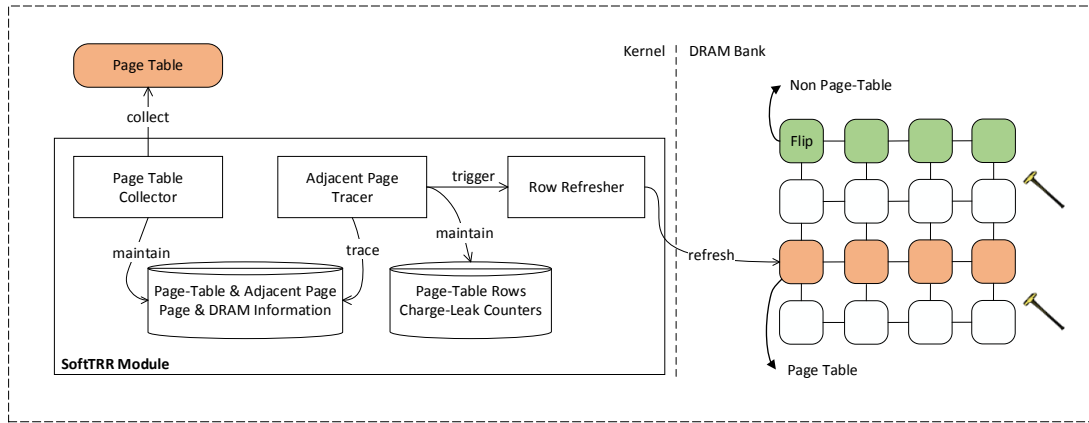


Figure 1: SoftTRR Overview. SoftTRR is a kernel module and has three main components. *Page Table collector* maintains information about page-table pages and their adjacent pages in close proximity. *Adjacent Page Tracer* traces access to the maintained adjacent pages and updates charge-leak counters for relevant rows of page-table pages. When the counters reach a pre-determined limit, *Row Refresher* is triggered to refresh desired rows hosting page-table pages. In comparison, non-page-table rows (highlighted in green) are vulnerable to bit flips.

3.2 Design Principles

SoftTRR follows the security and practicality design principles described below. The security principle is to guarantee SoftTRR can defend against all existing rowhammer attacks targeting page tables. The practicality principles aim to make SoftTRR applicable to real-world systems.

- **DP1:** SoftTRR should be effective in protecting ALL page tables. Without this completeness guarantee, an attacker can gain kernel privilege by compromising the integrity of page tables that are not protected by SoftTRR.
- **DP2:** SoftTRR should be compatible with OS kernels. It neither modifies/adds kernel source code nor breaks kernel code integrity through binary instrumentation, which hinders its adoption in practice.
- **DP3:** SoftTRR should have small performance overhead to a protected system.

3.3 Design Overview

SoftTRR, residing in the kernel space, collects all page tables, and monitors their entire life cycle from page-table creation to page-table release. For each collected page-table page, SoftTRR identifies all its adjacent pages in DRAM and traces memory accesses to the adjacent pages. Thus, SoftTRR is aware of which adjacent page is accessed. When the traced access count reaches a pre-determined limit, SoftTRR knows which page-table page is at the risk of being flipped and promptly refreshes the page (satisfying **DP1**).

All existing software-only mitigation techniques (see [Section 2](#)) deeply hack into the memory allocator to become

DRAM-aware and add extra allocation/deallocation constraints. Unlike them, SoftTRR only acquires offline domain knowledge (e.g., DRAM address (re)mappings of physical addresses), without requiring a new memory allocator or changing legacy allocator logic (satisfying **DP2**).

When paging is enabled, memory accesses are performed through page tables or relevant TLB entries, and SoftTRR flushes TLB and configures page tables to trace memory accesses to those adjacent pages. Thus, the access to an adjacent page raises a hardware exception, which is captured by SoftTRR for the tracing purpose. If no such access occurs, no overhead is introduced. Thus, the accesses to non-adjacent pages are at full speed, isolating the performance overhead caused by the accesses to adjacent pages (satisfying **DP3**).

As shown in [Figure 1](#), SoftTRR has three critical components. *Page Table Collector* actively collects all page tables and maintains their page and DRAM information. It also collects and maintains *adjacent pages*. Besides being accessible to unprivileged users, a page is considered as adjacent when itself or its corresponding page-table page is adjacent to (N-row from) another page-table page. This is based on an observation from Zhang et al. [62]. In particular, rowhammer attacks corrupting page tables are classified into two categories. For *explicit* attacks [13, 46], they require attacker-accessible memory adjacent to L1PT pages. For *implicit* attacks [62], they only need mutual adjacency among L1PT pages.

Adjacent Page Tracer keeps a close watch over memory accesses to collected adjacent pages, and maintains a charge-leak counter for a row where a page-table page resides. If any one row of adjacent pages has been accessed, the charge-leak counters of nearby page-table rows are updated accordingly, indicating that the page-table rows leak charge once.

Row Refresher remains dormant if charge-leak counters do not reach a pre-determined limit. If yes, a rowhammer attempt is believed to be taking place and the above tracer triggers row refresher, which will promptly refreshes desired rows whose charge-leak counters reach the limit.

4 Implementation

As stated in [Section 3.1](#), SoftTRR implements L1PT protection and a row of adjacent pages can be up to 6-row away from a row of L1PT pages. Our prototype implementation is a loadable kernel module (LKM) without modifications to the kernel. The LKM consists of around 1700 source lines of code and works with Ubuntu installation running a default Linux kernel 4.4.211. Before we talk about the three aforementioned components of SoftTRR, we first introduce important data structures as below.

4.1 Data Structures

We reuse the kernel's red-black tree structure [16], an efficient self-balancing binary search tree that guarantees searching in $\Theta(\log n)$ time (n is the number of tree nodes). As shown in [Table 1](#), we have three red-black trees and a ring buffer, i.e., `pt_rbtree`, `adj_rbtree`, `pt_row_rbtree` and `pte_ringbuf`, respectively.

Specifically, `pt_rbtree` stores L1PT page information while `adj_rbtree` stores information of pages that are adjacent to L1PT pages. For the two trees, a physical page number (PPN) is used as the node key and thus a new node will be allocated when information of a new L1PT page or adjacent page needs to be stored. Besides, `pt_row_rbtree` stores DRAM information about L1PT pages. For this tree node, `row_index` works as the node key and a node can have one or more bank structures (i.e., `bank_struct`). One bank structure stores `bank_index` that one or more L1PT pages own (e.g., multiple L1PT pages share the same row of the same bank). Also note that a page can span across multiple banks [55] and thus an L1PT page can have multiple `bank_struct`. `pt_count` records the number of L1PT PPNs that are in the same row of the same bank. `leak_count`, short for the charge-leak counter in [Section 3.3](#), stores the number of accesses to rows that are adjacent to a row of `row_index` in the same bank. For a given DRAM module, we leverage a publicly available algorithm [55] to reverse-engineer its DRAM address mapping, and embed the mapping into the kernel before acquiring a physical page's DRAM information. We allocate each node of each tree using the slab allocator [10], which is an efficient memory management mechanism intended for the kernel's small object allocation compared to the buddy allocator.

`pte_ringbuf` stores information of leaf page table entries (PTEs) that are collected by adjacent page tracer (see [Section 4.3](#)). These PTEs point to either adjacent pages them-

selves or *huge pages* containing adjacent pages. If the adjacent page is a 4 KiB page, the PTE is an L1PT entry. If the adjacent page is part of a huge page (i.e., 2 MiB or 1 GiB), the PTE is either an L2PT entry or an L3PT entry. Each node of `pte_ringbuf` is a structure that has three main fields also shown in [Table 1](#). Particularly, `pte` is a pointer to the leaf PTE. `vaddr` is a virtual address referring to an adjacent page or its corresponding huge page. `mm` is a pointer to a kernel structure (i.e., `mm_struct`) about a process's address space where `vaddr` belongs. The adjacent page tracer combines `vaddr` and `mm` to flush the TLB entry that stores the adjacent page's virtual-to-physical address mapping.

4.2 Page Table Collector

For user processes/threads that are already in the main memory before our module is loaded, page table collector enumerates the list of `task_struct` to find every existing process/thread, as Linux kernel uses `task_struct` for existing user processes/threads. It then performs page-table walk for every virtual page in each valid virtual memory area (VMA) of each user process to collect information of L1PT pages and their adjacent pages. Specifically, `pt_rbtree` and `pt_row_rbtree` store distinct L1PT pages, and their DRAM bank and row indexes, respectively. To build `adj_rbtree`, the collector finds out all user pages that are adjacent to L1PT pages in DRAM. It also selects all L1PT pages from `pt_rbtree` that are adjacent to each other and puts all PPNs pointed by selected L1PT pages' valid entries into `adj_rbtree`. For free pages that are adjacent to L1PT pages and allocated for use later (e.g., a free page is allocated and mapped to the user space right after the collector finishes collecting all adjacent pages), the adjacent page tracer handles them appropriately (see [Section 4.3](#)).

For L1PT pages that are dynamically allocated or freed after the above collection, we perform dynamic inline hooks to multiple kernel functions. Inline hook is called trampoline or detours hook, which is a method of receiving control when a hooked function is called. Dynamic kernel hook only requires loading a kernel module without kernel recompilation or binary rewriting, making itself easy to deploy in practice (e.g., Kprobes, Kpatch [19, 35, 36]).

We leverage a library¹ to hook two kernel functions, i.e., `__pte_alloc` and `__free_pages`. `__pte_alloc` traces newly allocated L1PT pages. `__free_pages` monitors dynamically released pages. The collector hooks these two functions to update the three red-black trees as follows:

- For a newly allocated L1PT page, its page, bank and row indexes will be updated into `pt_rbtree` and `pt_row_rbtree`, respectively. If there are new user pages that are adjacent to the L1PT page, they are added into `adj_rbtree`.

¹https://github.com/cppcoffee/inl_hook

Data Structures	Main Fields in A Node	Descriptions	
pt_rbtrees	PPN (key)	A unique page frame number of an LIPT page.	
adj_rbtrees	PPN (key)	A unique page frame number of an adjacent page.	
	row_index (key)	A row index of one or more LIPT pages.	
pt_row_rbtrees	bank_struct	bank_index	A bank index of one or more LIPT pages.
		pt_count	The number of LIPT pages that have the same indexes of bank and row.
		leak_count	The number of accesses to rows adjacent to a row of row_index and bank_index.
	pte	A pointer to a page table entry relevant to an adjacent page.	
pte_ringbuf	vaddr	A virtual address relevant to an adjacent page.	
	mm	A pointer to mm_struct relevant to a process where vaddr resides.	

Table 1: Data structures used by SoftTRR.

- If an adjacent page is freed, it will be removed from `adj_rbtrees`.
- If an LIPT page is freed, it will be removed from `pt_rbtrees`. Also, the collector acquires a node in `pt_row_rbtrees` that has the freed page's row index. Within the node, `pt_count` in each `bank_struct` corresponding to the freed page is decremented by one. If every `pt_count` for the node becomes 0, then the node is deleted from `pt_row_rbtrees`. Besides, the freed page's adjacent pages in `adj_rbtrees` are removed.

4.3 Adjacent Page Tracer

To trace memory accesses to adjacent pages at runtime, the adjacent page tracer leverages page fault handler.

Page Fault Handler. A page fault is a type of hardware exception. Whenever a user access to a virtual page violates access permissions dictated by one PTE, a page fault arises and will be captured by the MMU. As a response, the MMU will switch the process context to the kernel, which invokes the page fault handler to handle the fault based on an error code. The error code is generated by hardware and there are 7 page-fault error codes [27]. For instance, when a memory access to a virtual address that is marked as non-present in the PTE (i.e., `present` bit is cleared), the access triggers a non-present page fault with `P` bit in the error code set to 0. To handle this page fault, the page fault handler can allocate a new physical page for the virtual address and marks the address as present in the PTE, the so-called *demand paging*.

Leverage Page Fault. The adjacent page tracer can trace the memory access to a page by configuring flag bits in a PTE and hooking the page fault handler (i.e., `do_page_fault` function in the kernel space). As the memory access can be *read*, *write* or *instruction fetch*, not every flag bit can be leveraged. For instance, a physical page becomes read-only when its corresponding PTE has `RW` bit cleared. Once write-access to the page occurs, a page fault is generated with `W/R` bit of the error code set to 1. Thus, we experimented with each flag bit, results of which show that both `present` bit and `rsrv` bit in a PTE can be used for the tracing purpose. Next, we discuss why the tracer chooses `rsrv` bit rather than `present` bit.

Particularly, configuring `present` bit to trace the memory

access causes a kernel crash, as the kernel performs active checks of `present` bit in a leaf PTE in multiple cases. For instance, when a process is forking a new child process, the kernel checks `present` bit in the process's leaf PTEs. If one of the PTEs points to a physical page that is traced, `present` bit in the PTE is set to 0 by the tracer. When such a case occurs to the kernel check, the kernel will abort, because the tracer is unaware of when the forking occurs and it cannot restore `present` bit to 1 to pass the kernel check.

On top of that, we observe that one PTE has multiple `rsrv` bits in x86 which are unused and set to 0 by default. An access to a page with one `rsrv` bit in the PTE set to 1 will trigger a page fault and generate an error code with `RSVD` bit set to 1 (this `RSVD` error has been leveraged in prior works [2, 6, 8, 18, 56] for different purposes). In contrast to the `present` bit check, the kernel does not check against leaf PTEs' `rsrv` bits. For instance, if an adjacent page is a part of a huge page of 2 MiB, its leaf PTE is an L2PT entry and the kernel does not inspect any `rsrv` bit in the entry. As the page table management is a core component of the kernel, its code logic remains relatively stable. Take a recent stable Linux kernel version (i.e., 5.10.4) as an example, there is no check against any `rsrv` bit, either. It is probably because that `rsrv` bits remain unused in leaf PTEs. In our implementation, the tracer chooses a `rsrv` bit, i.e., bit 51 in the PTE.

Trace Adjacent Page. Upon the tracer has configured `rsrv` bits in relevant PTEs pointing to the adjacent pages or the huge pages containing the adjacent pages, and flushed desired TLB entries, subsequent access to an adjacent page or its huge page will trigger a page fault. As `do_page_fault` is hooked, the tracer captures a faulting (huge) page with an expected error code of `RSVD` and collects complete DRAM information from the faulting (huge) page. Thus, the tracer updates `leak_count` of LIPT pages that are adjacent to either the captured (huge) page or its leaf page-table page. As an LIPT page may have multiple `bank_struct`, `leak_count` of each `bank_struct` for the LIPT page should be updated accordingly. If the `leak_count` reaches a pre-determined limit in Figure 2, row refresher will be triggered (see Section 4.4).

We note that the tracer clears `rsrv` bit before transferring control back to the user space to resume the memory access. However, any subsequent access to the same adjacent page

or its huge page is no longer traced as `rsrv` bit is cleared. To address this issue, the tracer sets up a periodic timer to configure `rsrv` bit in a fixed interval and thus traces the accesses as frequently as possible. Specifically, when a timer comes, the tracer leverages kernel's *reverse mapping* feature to translate a PPN in `adj_rbtree` to a set of virtual addresses, as a PPN can be mapped to multiple virtual addresses. For each address, the tracer performs page-table walk, sets `rsrv` bit in its leaf PTE and flushes its cached TLB entry.

It is clearly inefficient to do the reverse-mapping and page-table walk for every PPN in `adj_rbtree` in every timer event. To improve the efficiency, the tracer sets `rsrv` bit in PTEs relevant to the pages in `adj_rbtree` and then frees corresponding nodes in `adj_rbtree` in the first timer. If page faults with the error code of `RSVD` occur, the tracer captures them and stores the faulting addresses' PTE information into a dedicated ring buffer (i.e., `pte_ringbuf`). When subsequent timer events come, the tracer sets `rsrv` bits in PTEs stored in `pte_ringbuf`, and handles remaining nodes in `adj_rbtree` which are updated by the page table collector.

For any new page that is allocated for the user space in the default page fault handler, the tracer checks if its PPN or its LIPT page's PPN (if exists) is adjacent to any PPN in `pt_rbtree`. If so, its leaf PTE information is inserted into `pte_ringbuf`.

Particularly, `pte_ringbuf` maintains two pointers for updates, i.e., `head` and `tail`. If a new PTE is inserted to `pte_ringbuf`, the `head` pointer is updated and points to the empty node next to the node of latest inserted PTE. If one PTE is removed from `pte_ringbuf` (i.e., its `rsrv` bit has been configured), the `tail` pointer is updated and points to the least recently inserted PTE. When the `head` and the `tail` point to the same ring buffer node, the buffer becomes empty. The ring buffer size is pre-determined empirically. When the node number between the `tail` and the `head` pointers is no less than 80% of the total node number of the ring buffer, the tracer allocates a larger ring buffer (e.g., four times of the old ring buffer size in our implementation), which will store newly inserted PTE. The old ring buffer will be freed when its stored PTEs are all consumed by the tracer.

As shown in [Figure 2](#), the time interval between two consecutive timer events (denoted as `timer_inr`) should be small enough to keep adjacent pages under close surveillance and `leak_count` is updated promptly. On the other hand, our system might experience unacceptable overhead if the timer is too frequent and causes numerous context switches between user and kernel. To this end, we discuss how to decide `timer_inr` in [Section 4.5](#) to keep SoftTRR's security guarantee while minimize its performance impacts.

4.4 Row Refresher

Direct-physical Map. Linux systems and paravirtualized hypervisors (e.g., Xen) map the whole available physical mem-

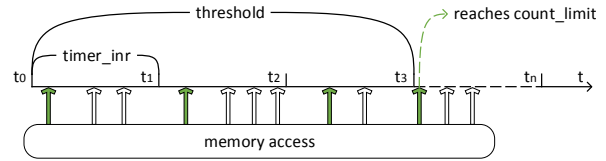


Figure 2: The adjacent page tracer sets up tracing to adjacent pages in each time point from $t_0, t_1, t_2, t_3, \dots, t_n$ and the interval between two adjacent time points is `timer_inr`. The tracer captures the first memory access (highlighted in green) and ignores subsequent memory accesses in each interval of `timer_inr` and updates `leak_count`. Whenever `leak_count` reaches `count_limit`, the row refresher starts.

ory directly into the kernel space [31, 58] in order for the kernel to access any data or code in the physical memory. Thus, every physical page allocated for the user space has been mapped to at least two virtual pages, i.e., a user virtual page and a kernel virtual page. While for a kernel's physical page, it is mapped to a single kernel virtual page.

Refresh Desired Rows. If `leak_count` in `bank_struct` reaches a pre-determined limit (denoted as `count_limit`), the row refresher refreshes desired rows specified by relevant `bank_struct`. As each node in `pt_row_rbtree` provides bank indexes and row indexes, the refresher leverages them to reconstruct a physical address. Based on the direct-physical map, the refresher finds out a kernel virtual address mapped to the physical address. As a read-access to a row can automatically re-charge the row and prevent potential bit flips, the refresher flushes CPU caches of the kernel virtual address, reads the virtual address, and resets `leak_count` to 0 at last.

If `count_limit` is set too small (e.g., 1), the refreshing cost may become unacceptable as many unnecessary refreshes are introduced by regular memory accesses to adjacent pages. If `count_limit` is too large, the refresher is unable to promptly refresh a row before it is flipped. Thus, `count_limit` should be no less than 2 and we decide its value in the next section.

4.5 Offline Profile

SoftTRR decides realistic and reasonable `timer_inr` and `count_limit` to keep its security and practicality design principles. As illustrated in [Figure 2](#), the adjacent page tracer only captures the first memory access to an adjacent page within each `timer_inr` and updates `leak_count`. The subsequent memory accesses within `timer_inr` to the same page will be ignored by the tracer. Thus, the maximum time period (denoted as `threshold`) for hammer before the page is refreshed has such an equation: $threshold = timer_inr \times (count_limit - 1)$. This means that SoftTRR must carefully set `threshold` short enough to ensure that no bit flip occurs within `threshold`.

We decide `threshold` based on the equation:

Machine Model	Hardware Configuration			Attack	SoftTRR Bit Flip Failed?
	CPU Arch.	CPU Model	DRAM (Part No.)		
Dell Optiplex 390	KabyLake	i7-7700k	Kingston DDR4 (99P5701-005.A00G)	Memory Spray [46]	✓
Dell Optiplex 990	SandyBridge	i5-2400	Samsung DDR3 (M378B5273DH0-CH9)	CATTmew [13]	✓
Thinkpad X230	IvyBridge	i5-3230M	Samsung DDR3 (M471B5273DH0-CH9)	PThammer [62]	✓

Table 2: Each rowhammer attack targets n (e.g., 50 in our experiments) victim pages of L1PTEs. With SoftTRR enabled, each attack fails to induce bit flips in these pages, indicating that those attacks have been mitigated.

$threshold = t_{RC} \times \#ACT$, where t_{RC} is the time interval between two successive ACT commands and $\#ACT$ is the number of activations for all the hammered rows that is required to induce the first bit flip. Thus, we guarantee that no bit flip occurs within the time interval of $threshold$. We learn from Kim et al. [32] that t_{RC} is around 50 ns and $\#ACT$ per row is in the order of 20 K on DDR3 modules and 10 K on DDR4 modules. Compared to DDR3 modules that require at least 1 aggressor row, no less than 2 aggressor rows are required in DDR4 modules due to the ChipTRR. As such, $\#ACT$ for triggering the first bit flip is around 20 K for both DDR3 and DDR4 modules. To this end, $threshold$ is set to 1 ms, below which DRAM modules are believed to be rowhammer-free. As both $timer_inr$ and $count_limit$ for SoftTRR are unsigned integers, $timer_inr$ is set to 1 ms and $count_limit$ is set to 2.

5 Security Evaluation

We now turn to evaluate the security effectiveness of SoftTRR on three different hardware configurations, summarized in Table 2, all running Ubuntu.

We deploy SoftTRR into each system against one representative kernel privilege escalation attack, i.e., Memory Spray [46] that hammers user memory adjacent to L1PTEs, CATTmew [13] that hammers device driver buffer adjacent to L1PTEs, and PThammer [62] that implicitly hammers L1PTEs adjacent to other L1PTEs. Both Memory Spray and CATTmew are explicit rowhammer attacks with two different types of memory accessible to unprivileged users. PThammer is the only published implicit rowhammer attack.

5.1 Defeating Memory Spray

Background. The Memory Spray [46] is the first rowhammer attack targeting L1PTEs. It is a probabilistic attack, as it sprays numerous L1PT pages into the memory with the hope that some L1PT pages are placed onto victim rows adjacent to attacker-controlled rows. As such, exploitable bits in L1PTEs can be flipped, resulting in kernel privilege escalation.

Evaluation Details. We test the effectiveness of SoftTRR against the Memory Spray on the Dell Optiplex 390. In this

machine, traditional 2-sided hammer pattern cannot trigger any bit flip and instead we use the 3-sided hammer identified by TRRespass². We first conduct 3-sided hammer to randomly identify n (e.g., 50 in our evaluation) vulnerable pages that have reproducible bit flips, that is, a vulnerable page has at least one victim physical address (P_v) and hammering three aggressor addresses P_a , P_b and P_c will flip bits in P_v .

We then optimize the attack by using the kernel privilege to put page tables onto vulnerable pages in a deterministic way. Specifically, we spray n pages of L1PTs by creating a virtual memory region of $2n$ MiB, ask the kernel to copy the content of the n pages of L1PTs into the n vulnerable pages, which are then used to translate the virtual memory region. The vulnerable pages now contain L1PTs and the original L1PTs are removed. By doing so, an attacker will definitely corrupt any one of the L1PTs pages by hammering three relevant aggressor addresses. When SoftTRR is enabled to collect and protect the n pages of L1PTs, we re-start the optimized attack for n hours (one-hour hammer for one vulnerable L1PT page) and observe no single bit flip in those n pages of L1PTs by checking their integrity, indicating that the Memory Spray attack has been successfully defeated.

5.2 Defeating CATTmew

Background. As mentioned in Section 2, CATT [12] enforces physical user-kernel isolation. CATTmew [13] breaks CATT’s security guarantee by identifying device (e.g., SCSI Generic) driver buffers that are kernel memory but can be accessed by unprivileged users. CATTmew exploits the driver buffers to ambush adjacent L1PT pages for hammer, with the hope that these L1PT pages are prone to bit flips.

Evaluation Details. We use 2-sided hammer to search n vulnerable pages on the Dell Optiplex 990. A vulnerable page has at least one victim physical address (P_v) and hammering two aggressor addresses (P_a and P_b) flips bits in P_v .

We then rely on the kernel privilege to convert CATTmew into a deterministic attack. Specifically, we spray n L1PT pages and copy their entries onto the n vulnerable pages as what we did in the optimized Memory Spray attack. On top of that, we apply for the SCSI Generic (SG) buffer using

²<https://github.com/vusec/trrespass>

Linux user APIs. In this test machine, we can apply as large as 123 MiB and only 8n KiB of the SG buffer are enough. We instruct the kernel to copy the allocated SG buffer’s content into the 2n aggressor pages and change the buffer’s address mappings accordingly. To this end, hammering the buffer will induce bit flips in the vulnerable L1PT pages. However, when SoftTRR is set active, no single bit flip has been observed in those L1PT pages after n hours of hammering, indicating that SoftTRR is effective in defeating the CATTmew attack.

5.3 Defeating PThammer

Background. Rowhammer attacks before PThammer [62] are explicit rowhammer that require access to an exploitable aggressor row (e.g. adjacent to a row of L1PTs). PThammer voids this requirement. By spaying L1PT pages and placing some onto victim rows with a high probability, PThammer exploits page-table walk to produce frequent loads of some L1PTes from aggressor rows (i.e., “implicitly hammering L1PTes”), which will induce bit flips in other L1PTes in victim rows.

Evaluation Details. We optimize PThammer by using the kernel privilege to present a more efficient and deterministic attack on the Thinkpad X230. Specifically, PThammer uses eviction sets to flush TLB entries and CPU caches of desired L1PTes and user memory loads trigger the page-table walk to implicitly hammer the L1PTes. However, the eviction-based flush is probabilistic. In our test, the kernel assists PThammer in performing the flush through explicit instructions (i.e., `invlpg` for TLB flush and `clflush` for L1PTes flush). Thus, its hammer instruction sequence is kernel-assisted flush with a user memory load, which is less efficient than the aforementioned 2-sided hammer that applies `clflush` for user data flush. In such a case, we cannot use the traditional 2-sided hammer to identify vulnerable pages, as these pages may become non-flippable to the kernel-assisted hammer. To address this issue, we add a certain number of NOP (e.g., 180) instructions into the 2-sided hammer instruction sequence to meet the time cost taken by the kernel-assisted hammer. By doing so, n vulnerable pages of interest can be discovered.

As PThammer massages L1PTes onto vulnerable pages with a probability, we instead spray 3n L1PT pages by creating a virtual memory region of 6n MiB. We then ask the kernel to copy all entries of the L1PT pages into the n vulnerable pages and the 2n aggressor pages. The kernel then changes the address mappings of the created virtual memory region by using the new 3n L1PT pages. As such, the optimized PThammer successfully induces bit flips in the n vulnerable L1PT pages by using the kernel-assisted hammer against the 2n aggressor L1PT pages. In comparison, we enable SoftTRR before starting the optimized PThammer. As each 2 aggressor L1PT pages is adjacent to a vulnerable L1PT page in `pt_rbtrees`, SoftTRR traces memory accesses to the created virtual pages pointed by the L1PT page entries. Considering

Benchmarks	Programs	SoftTRR Overhead	
		$\Delta_{\pm 1}$	$\Delta_{\pm 6}$ (default)
SPECspeed 2017 Integer	perlbench_s	0.67%	0.67%
	gcc_s	0.23%	0.92%
	mcf_s	-0.76%	0.30%
	omnetpp_s	-0.81%	1.82%
	xalancbmk_s	0.36%	2.50%
	x264_s	0.00%	0.61%
	deepsjeng_s	0.00%	0.28%
	leela_s	0.23%	0.46%
	exchange2_s	-0.70%	-0.23%
	xz_s	1.48%	0.93%
	Mean	0.07%	0.83%
Phoronix	Apache	-0.16%	0.32%
	unpack-linux	1.31%	1.84%
	iozone	0.89%	-1.15%
	postmark	0.89%	0.00%
	stream:Copy	0.01%	0.00%
	stream:Scale	0.60%	0.23%
	stream:Triad	0.07%	0.37%
	stream:Add	0.03%	0.35%
	compress-7zip	1.52%	2.24%
	openssl	0.14%	0.13%
	pybench	0.00%	0.52%
	phpbench	0.92%	0.01%
	cacheben:read	-0.38%	0.26%
	cacheben:write	-0.26%	-0.44%
	cacheben:modify	-0.01%	0.67%
	ramspeed:INT	-0.09%	-0.63%
	ramspeed:FP	-0.15%	-0.63%
	Mean	0.22%	0.24%
memcached	Statistics		
	Ops	0.39%	0.18%
	TPS	0.39%	0.15%
	Net_rate	0.46%	0.31%

Table 3: Benchmark results for SPECspeed 2017 Integer, Phoronix and memcached.

that the PThammer still requires frequent memory loads of the created virtual pages for page-table walk, it cannot bypass the tracing. After n hours of hammering the 2n aggressor pages, no bit flip occurs, meaning that SoftTRR has mitigated PThammer.

6 Performance Evaluation

We evaluate the performance impacts induced by SoftTRR, i.e., SoftTRR’s runtime overhead, memory consumption and system robustness are evaluated in Section 6.1, Section 6.2 and Section 6.3, respectively. The experiments are conducted in a DDR4-based system. The system is Ubuntu running on top of a Dell Desktop with Intel i7-7700K and Samsung 16 GiB DDR4 (part number: M378A2G43AB3-CWE). By default, the row distance implemented by SoftTRR between adjacent rows and L1PT-page rows is up to 6-row, denoted by $\Delta_{\pm 6}$. In comparison, we also measure its impacts in the scenario of only one-row-distance that previous works (e.g., [34]) assume, denoted by $\Delta_{\pm 1}$. The results show that SoftTRR

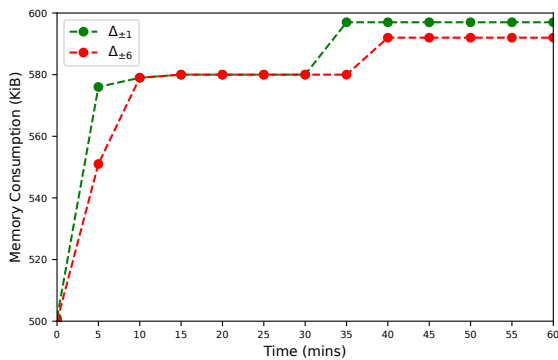


Figure 3: The memory consumed by SoftTRR in both $\Delta_{\pm 1}$ and $\Delta_{\pm 6}$ for the LAMP production environment.

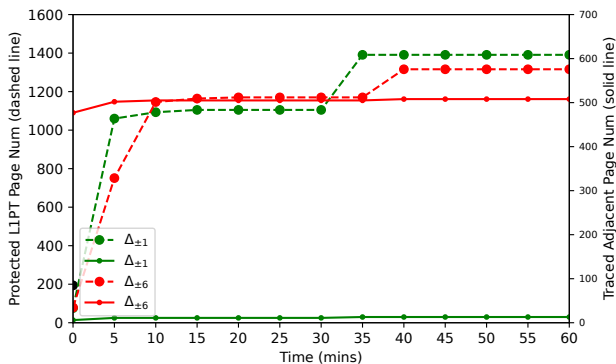


Figure 4: The numbers of protected LIPT pages and traced adjacent pages in both $\Delta_{\pm 1}$ and $\Delta_{\pm 6}$ for the LAMP production environment.

in both scenarios of $\Delta_{\pm 6}$ and $\Delta_{\pm 1}$ incurs an average slow-down within 0.83% indicating that the row distance may have a relatively small impact on the performance overhead. We note that the cost of initially loading SoftTRR into the kernel is around 28 ms and it occurs only once. We also validate the system robustness of SoftTRR, results of which show that SoftTRR does not affect the stability of the protected system, making itself practical.

6.1 Benchmark Runtime Overhead

We measure SoftTRR-induced runtime overhead using two popular benchmarks and an industrial memory-intensive application, i.e., SPECspeed 2017 Integer [50], Phoronix test suite³ and memcached⁴.

³<https://github.com/phoronix-test-suite/phoronix-test-suite>

⁴<http://memcached.org/latest>

SPEC CPU 2017 is an industry standard benchmark package that contains CPU-intensive programs for measuring compute-intensive performance. It has 43 benchmarks in total and is organized into 4 suites, among which SPECspeed 2017 Integer has been used. This suite launches 10 integer programs with a specific configuration file customized from *Example-linux-gcc-x86.cfg* and the benchmark results are summarized in Table 3. As we can see from the table, the overhead of $\Delta_{\pm 6}$ (0.83%) and $\Delta_{\pm 1}$ (0.07%) are less than 1%.

Phoronix is a free and open-source benchmark software for mainstream OSes (e.g., Linux, MacOS and Windows). It allows for testing performance overhead against common applications in an automated manner. As this suite has a large number of programs testing different aspects of a system, we select a subset of the available programs to stress-test performance of CPU, memory, network I/O and disk I/O. As shown in Table 3, the average performance overhead is 0.22% for $\Delta_{\pm 1}$ and 0.24% for $\Delta_{\pm 6}$, respectively, indicating that the Phoronix overhead is negligible in both scenarios.

memcached is a pervasively used in-memory data storage system and can consume as much memory as possible. To evaluate the performance impacts of SoftTRR on memcached, we start memcached as a memory-intensive process, that is, 13 out of 16 GiB are allocated for memcached, to stress-test SoftTRR. We then run memaslap [1] for 5 times (with 5 minutes in each time) to benchmark the memcached process. The memaslap tool is a load generation and benchmark for memcached-based servers and allows generating various workloads. In our experiments, memaslap specifies default workloads for memcached (i.e., the task window size is 10 K, the thread for startup is 1 and each thread has 16 self-governed concurrencies to handle socket connections). As shown in Table 3, the average overhead of Ops, TPS and Net_rate are only 0.39%, 0.39% and 0.46% for $\Delta_{\pm 1}$ and 0.18%, 0.15%, and 0.31% for $\Delta_{\pm 6}$, respectively.

6.2 LAMP Runtime Memory Consumption

We use a real-world use case to measure runtime memory consumption of SoftTRR, that is, a LAMP server (i.e., Linux, Apache, MySQL and PHP). We run a common tool (i.e., Nikto [51]) in another machine for 60 minutes to stress test the LAMP server. Nikto is a web server scanner that tests the LAMP server for insecure files and outdated server software. It also carries out generic and server type specific checks.

The memory cost induced by SoftTRR within the 60 minutes is shown in Figure 3. The memory consumption is a total memory size of three red-black trees (i.e., pt_rbtrees, pt_row_rbtrees and adj_rbtrees) and the ring buffer (i.e., pte_ringbuf). We note that the pre-allocated pte_ringbuf is 396 KiB. As shown in the figure, the memory costs in both $\Delta_{\pm 1}$ and $\Delta_{\pm 6}$ increase gradually and reach a relatively stable level in the last 15 minutes. Both $\Delta_{\pm 1}$ and $\Delta_{\pm 6}$ have a similar and low memory cost (i.e., less than 600 KiB).

Linux Test Project	Vanilla System	SoftTRR	
		$\Delta_{\pm 1}$	$\Delta_{\pm 6}$ (default)
File	open	✓	✓
	close	✓	✓
	ftruncate	✓	✓
	rename	✓	✓
Network	Listen	✓	✓
	Socket	✓	✓
	Send	✓	✓
	Recv	✓	✓
Memory	mmap	✓	✓
	munmap	✓	✓
	brk	✓	✓
	mlock	✓	✓
	munlock	✓	✓
	mremap	✓	✓
Process	getpid	✓	✓
	exit	✓	✓
	clone	✓	✓
Misc.	ioctl	✓	✓
	prctl	✓	✓
	vhangup	✓	✓

Table 4: System-call stress tests from Linux Test Project (✓: the stress test does not report any problem.).

Protected and Traced Page Number. When computing the memory consumption, we also collect the unique page numbers that SoftTRR protects and traces, respectively. Figure 4 shows that both protected L1PT page number and traced adjacent page number in $\Delta_{\pm 1}$ and $\Delta_{\pm 6}$ increase and become stable within the 60 minutes. We note that the protected L1PT page numbers in $\Delta_{\pm 1}$ and $\Delta_{\pm 6}$ are in the same order of magnitude as the overall system activities in both scenarios are similar to each other. As an L1PT-page row in $\Delta_{\pm 6}$ can have up to 12 adjacent rows, 6 times the adjacent row number that an L1PT-page row can have in $\Delta_{\pm 1}$, more adjacent pages are expected to be collected in $\Delta_{\pm 6}$. Figure 4 shows that the traced adjacent page number in $\Delta_{\pm 6}$ is higher than that in $\Delta_{\pm 1}$.

6.3 System Robustness

To evaluate the robustness of our test system after deploying SoftTRR, we select 20 system calls of different types and perform stress tests for each selected system call on both the vanilla system and the SoftTRR-based system. The stress tests come from Linux Test Project (LTP)⁵ and they are used to identify system problems. Particularly, we follow the LTP’s quick guide to run a single test each time using the default configuration without explicitly specifying any parameters (e.g., binding the test onto one or more CPU cores. As such, we first run all the tests on the vanilla system, results of which are used as the baseline to compare with that of SoftTRR. As can be seen from Table 4, the stress test results clearly

⁵<https://github.com/linux-test-project/ltp>

show that there is no deviation for the SoftTRR-based system compared to the vanilla system. Besides, we do not observe any issue when executing previous benchmarks. As a result, the test system runs stably with SoftTRR enabled.

7 Discussion

Other Data Objects Protection. If critical data structures of SoftTRR are targeted, we can easily extend SoftTRR to protect them. Similar to the L1PT protection described in Section 3.3, SoftTRR treats its own data structures as protected objects. To protect sensitive user objects (e.g., binary code pages of `setuid` processes or DNN model weight pages) against existing attacks [21, 24], RIP-RH [9] is effective by physically isolating trusted user processes. Orthogonal to RIP-RH, SoftTRR can also be extended to defeat such attacks. Particularly, trusted users pass specified objects to SoftTRR through a provided user API (e.g., `netlink`) and SoftTRR thus uses a similar mechanism to protect those objects.

Level-1 and Higher-level Page Table. Existing kernel privilege escalation attacks focus on corrupting L1PTs, and there is no demonstrated attack that has successfully exploited higher-level page tables [57]. If such an attack may be feasible in the future, we can easily extend our SoftTRR to protect higher-level page tables. For instance, when SoftTRR is extended to protect L2PT pages, SoftTRR collects desired user pages if they or their corresponding L1PT or L2PT pages are adjacent to either L1PT or L2PT pages. SoftTRR traces the collected user pages by setting `rsrv` bits in their leaf PTEs and refreshes relevant page-table pages when necessary. As the number of higher-level PT pages is significantly smaller than the number of L1PT pages (e.g., an L2PT page can point up to 512 L1PT pages), we believe that the additional performance overhead will not be high.

DMA-based Kernel Privilege Escalation Attack. There is NO existing DMA-based kernel privilege escalation attack on x86. Such attack is demonstrated on ARM (Drammer [53]), and it has been defeated by GuardION [54] that enforces DMA memory isolation. In the future, if such attacks on x86 prove to be feasible, we can take the following two ways to solve. One is to integrate SoftTRR with existing orthogonal defenses. In particular, ALIS [52] on x86 physically isolates DMA memory using guard rows and bit flips are thus confined to DMA memory of attackers.

Alternatively, SoftTRR can leverage IOMMU [26] to monitor remote access to DMA memory by configuring I/O page tables, similar to MMU-based page tables. Specifically, SoftTRR collects (I/O) page tables and their adjacent DMA memory pages that are allocated to users. By configuring I/O page tables, SoftTRR traces accesses to the collected DMA pages. When IOMMU is widely available on x86, we believe that SoftTRR can leverage it to defend (I/O) page tables against unknown DMA-based kernel privilege escalation attacks.

Half-Double Attack. Inspired by [17], Google recently proposes a new hammer technique, called Half-Double [20], which induces bit flips in a target victim row that is 2-row away from a row being hammered. Specifically, Half-Double observes that some ChipTRR implementations in DDR4 modules will refresh a row’s two neighboring rows if the row is detected to be hammered. With this key observation, Half-Double hammers a row (known as Far Aggressor), which enables ChipTRR to frequently refresh the row’s neighboring rows (known as Near Aggressor). As such, Half-Double can combine the frequent refreshes and a few activations against Near Aggressors to induce bit flips in victim rows that are 2-row away from a Far Aggressor.

However, we believe that Half-Double is unlikely to bypass SoftTRR and break page tables. In order to induce the first bit flip, #ACT required by Half-Double to hammer one Far Aggressor is about 296K, whereas SoftTRR assumes that the minimum #ACT is 20K for the first bit flip based on Kim et al. [32]. Thus, SoftTRR can detect Half-Double’s hammering and refresh page-table rows (if any) from being flipped by a 2-row-distance Far Aggressor. In the current implementation of SoftTRR, it can protect page-table rows from being corrupted by a Far Aggressor that is up to 6-row away.

Possible Performance Degradation Attack. We did not observe a high performance impact in our real-world applications and it might rarely occur that memory accesses concentrate on locations adjacent to LIPTEs. The system performance can be badly affected (as a kind of DoS attack [41]) if an adversary stresses SoftTRR by causing many additional page faults and refreshes. To alleviate such attack, SoftTRR can count the number of refreshes. If the count reaches a threshold, it can raise an alarm and leverage the scheduling information to narrow down the list of potentially malicious processes.

Support for ARM Architecture. Although there are reserved bits in page table entries in the ARM architecture, setting these bits will not trigger any hardware fault [4]. If we extend SoftTRR to provide ARM support, a possible solution is to disable the page table walk and capture the address-translation fault. However, this solution may introduce a larger performance overhead, as each memory access to a process triggers the fault if the process has pages adjacent to LIPT pages. Alternatively, we can leverage the present bit rather than the reserved bit in both x86 and ARM. As discussed in Section 4.3, the kernel performs active checks of the present bit in a leaf PTE. To address this issue, we can leverage the approach [56] to find all the functions where the kernel performs the check. By hooking these functions, we can restore the present bit and bypass the kernel check.

Support for Hardware-assisted Virtualization . SoftTRR, by design, works in a bare-metal system. To adapt itself to work in the guest OS kernel of a VM, SoftTRR needs the mappings of guest physical addresses to DRAM addresses. As

host physical addresses to DRAM mappings and in-DRAM address remappings can be reverse-engineered through prior works [14, 44, 55, 59], SoftTRR requires the guest-to-host memory mapping that is managed by the hypervisor. To this end, SoftTRR can register a virtual interrupt to communicate with the hypervisor. Particularly, upon the VM’s physical memory is allocated, SoftTRR obtains the guest-to-host memory mapping through the registered interrupt. If the VM’s physical memory is updated at runtime, the hypervisor notifies the SoftTRR of the updated mapping. If the hypervisor maintains mostly consecutive mapping (e.g., the Xen hypervisor uses 1 GiB huge-pages by default), we do not think maintaining the mapping would cause a major issue (e.g., SoftTRR only maintains a mapping of 1K entries even if the VM’s memory is up to 1 TiB).

8 Conclusion

In this paper, we proposed a software-only defense, named SoftTRR, that protects level-1 page tables against rowhammer attacks on x86. SoftTRR is a loadable kernel module and compatible with commodity Linux systems without requiring any kernel modification.

We evaluated the security effectiveness of SoftTRR-enabled systems using three kernel privilege escalation attacks. Also, we measured SoftTRR’s performance overhead, memory cost, and stability using multiple benchmark suites and a real-world use case. The experimental results indicate that SoftTRR is effective in defending against all the mentioned attacks, and practical in incurring low performance overhead and memory cost. Besides, it does not affect the system stability.

Acknowledgments

We thank our anonymous reviewers and shepherd for their insightful comments and suggestions. This work was supported in part by the National Key R&D Program of China (Grant No. 2020YFB1805402), the National Natural Science Foundation of China (Grant No. 61802397, 62002167, 61901209, U1736208 and 61902374), and the National Natural Science Foundation of JiangSu (Grant No. BK20200461). Zhi Zhang and Yueqiang Cheng are joint first authors. Wenhao Wang is the corresponding author.

References

- [1] A Load Generation and Benchmark Tool. memaslap. <http://docs.libmemcached.org/bin/memaslap.html>.
- [2] Neha Agarwal and Thomas F Wenisch. Thermo-stat: Application-transparent page management for two-

- tiered main memory. In *Architectural Support for Programming Languages and Operating Systems*, pages 631–644, 2017.
- [3] Apple, Inc. About the security content of mac efi security update 2015-001. <https://support.apple.com/en-au/HT204934>, August 2015.
- [4] ARM, Inc. Arm architecture reference manual armv8, for armv8-a architecture profile. <https://developer.arm.com/documentation/ddi0487/gb>.
- [5] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-based protection against next-generation rowhammer attacks. In *Architectural Support for Programming Languages and Operating Systems*, pages 743–755, 2016.
- [6] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. Efficient virtual memory for big memory servers. In *International Symposium on Computer Architecture*, pages 237–248, 2013.
- [7] Tanj Bennett, Stefan Saroiu, Alec Wolman, and Lucian Cojocar. Panopticon: A complete in-dram rowhammer mitigation. In *Workshop on DRAM Security*, 2021.
- [8] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *International Symposium on Microarchitecture*, pages 383–394, 2013.
- [9] Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahamd-Reza Sadeghi. RIP-RH: Preventing rowhammer-based inter-process attacks. In *Asia Conference on Computer and Communications Security*, pages 561–572, 2019.
- [10] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, 1994.
- [11] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: memory deduplication as an advanced exploitation vector. In *IEEE Symposium on Security and Privacy*, pages 987–1004, 2016.
- [12] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAN't Touch This: Software-only mitigation against rowhammer attacks targeting kernel memory. In *USENIX Security Symposium*, 2017.
- [13] Yueqiang Cheng, Zhi Zhang, Surya Nepal, and Zhi Wang. CATTmew: Defeating software-only physical kernel isolation. *IEEE Transactions on Dependable and Secure Computing*, 2019.
- [14] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are we susceptible to rowhammer? an end-to-end methodology for cloud providers. In *IEEE Symposium on Security and Privacy*, May 2020.
- [15] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: on the effectiveness of ECC memory against rowhammer attacks. In *IEEE Symposium on Security and Privacy*, pages 55–71, 2019.
- [16] Jonathan Corbet. Trees ii: red-black trees. <https://lwn.net/Articles/184495/>, 2006.
- [17] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the many sides of target row refresh. In *IEEE Symposium on Security and Privacy*, 2020.
- [18] Jayneel Gandhi, Arkaprava Basu, Mark D Hill, and Michael M Swift. Badgertrap: A tool to instrument x86-64 tlb misses. *ACM SIGARCH Computer Architecture News*, 42(2):20–23, 2014.
- [19] Mohamad Gebai and Michel R Dagenais. Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead. *ACM Computing Surveys*, pages 1–33, 2018.
- [20] Google, Inc. Half-double: Next-row-over assisted rowhammer. https://github.com/google/hammer-kit/blob/main/20210525_half_double.pdf, May 2021.
- [21] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O’Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *IEEE Symposium on Security and Privacy*, pages 245–261, 2018.
- [22] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321, 2016.
- [23] Hasan Hassan, Yahya Can Tugrul, Jeremie S Kim, Victor Van der Veen, Kaveh Razavi, and Onur Mutlu. Uncovering in-dram rowhammer protection mechanisms: A new methodology, custom rowhammer patterns, and implications. In *International Symposium on Microarchitecture*, pages 1198–1213, 2021.
- [24] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitras. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *USENIX Security Symposium*, pages 497–514, 2019.

- [25] HP, Inc. Hp moonshot component pack. <https://support.hpe.com/hpsc/doc/public/display?docId=c04676483>, May 2015.
- [26] Intel, Inc. Intel 64 and IA-32 architectures software developer’s manual combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c. October 2011.
- [27] Intel, Inc. Intel 64 and IA-32 architectures optimization reference manual. September 2014.
- [28] Intel, Inc. The role of ecc memory. <https://www.intel.com/content/www/us/en/workstations/workstation-ecc-memory-brief.html>, 2015.
- [29] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *IEEE Symposium on Security and Privacy*, 2022.
- [30] JEDEC Solid State Technology Association. Low power double data rate 4 (LPDDR4). <https://www.jedec.org/standards-documents/docs/jesd209-4b>, 2015.
- [31] Kernel.org. Virtual memory map with 4 level page tables (x86_64). https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt, 2009.
- [32] Jeremie S Kim, Minesh Patel, A Giray Yaglikci, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques. In *International Symposium on Computer Architecture*, 2020.
- [33] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. In *International Symposium on Computer Architecture*, page 361–372, 2014.
- [34] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriese, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. ZebRAM: comprehensive and compatible software protection against rowhammer attacks. In *Operating Systems Design and Implementation*, pages 697–710, 2018.
- [35] Anil Kurmus, Sergej Dechand, and Rüdiger Kapitza. Quantifiable run-time kernel attack surface reduction. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 212–234, 2014.
- [36] Anil Kurmus, Alessandro Sorniotti, and Rüdiger Kapitza. Attack surface reduction for commodity os kernels: trimmed garden plants may attract less bugs. In *Proceedings of the Fourth European Workshop on System Security*, pages 1–6, 2011.
- [37] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading bits in memory without accessing them. In *IEEE Symposium on Security and Privacy*, 2020.
- [38] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. TWiCe: preventing row-hammering by exploiting time window counters. In *International Symposium on Computer Architecture*, pages 385–396, 2019.
- [39] LENOVO, Inc. Row hammer privilege escalation lenovo security advisory. https://support.lenovo.com/au/en/product_security/row_hammer, August 2015.
- [40] Micron, Inc. DDR4 SDRAM Datasheet. <https://www.micron.com/products/dram/ddr4-sdram/>, 2015.
- [41] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security Symposium*, 2007.
- [42] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.
- [43] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. Graphene: Strong yet lightweight row hammer protection. In *International Symposium on Microarchitecture*, pages 1–13, 2020.
- [44] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security Symposium*, pages 565–581, 2016.
- [45] Mark Seaborn. How physical addresses map to rows and banks in dram. <http://lackingrhoticity.blogspot.com.au/2015/05/how-physical-addresses-map-to-rows-and-banks.html>, 2015.
- [46] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat’15*, 2015.
- [47] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. Counter-based tree structure for row hammering mitigation in DRAM. *IEEE Computer Architecture Letters*, 16(1):18–21, 2016.
- [48] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. Mitigating wordline crosstalk using adaptive trees of counters. In *International Symposium on Computer Architecture*, pages 612–623, 2018.

- [49] Mungyu Son, Hyunsun Park, Junwhan Ahn, and Sungjoo Yoo. Making DRAM stronger against row hammering. In *Design Automation Conference*, pages 1–6, 2017.
- [50] Standard Performance Evaluation Corporation. Spec cpu 2017. <https://www.spec.org>, 2017.
- [51] Chris Sullo. <https://cirt.net/nikto>, 2012.
- [52] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *USENIX Annual Technical Conference*, 2018.
- [53] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1675–1689, 2016.
- [54] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. Guardian: Practical mitigation of dma-based rowhammer attacks on arm. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 92–113. Springer, 2018.
- [55] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. Dramdig: A knowledge-assisted tool to uncover dram address mapping. In *Design Automation Conference*, 2020.
- [56] Zhe Wang, Chenggang Wu, Yinqian Zhang, Bowen Tang, Pen-Chung Yew, Mengyao Xie, Yuanming Lai, Yan Kang, Yueqiang Cheng, and Zhiping Shi. Safehidden: an efficient and secure information hiding technique using re-randomization. In *USENIX Security Symposium*, pages 1239–1256, 2019.
- [57] Xin-Chuan Wu, Timothy Sherwood, Frederic T. Chong, and Yanjing Li. Protecting page tables from rowhammer attacks using monotonic pointers in DRAM true-cells. In *Architectural Support for Programming Languages and Operating Systems*, pages 645–657, 2019.
- [58] xenbits.xen.org. source code (page.h). http://xenbits.xen.org/gitweb/?p=xen.git;a=blob;hb=refs/heads/stable-4.3;f=xen/include/asm-x86/x86_64/page.h, 2009.
- [59] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation. In *USENIX Security Symposium*, pages 19–35, 2016.
- [60] Abdullah Giray Yağlıkçı, Minesh Patel, Jeremie S Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Ghose Saugata, and Mutlu Onur. Blockhammer: Preventing rowhammer at low cost by blacklisting rapidly-accessed dram rows. In *High Performance Computer Architecture*, 2021.
- [61] Zhenkai Zhang, Zihao Zhan, Daniel Balasubramanian, Bo Li, Peter Volgyesi, and Xenofon Koutsoukos. Leveraging EM side-channel information to detect rowhammer attacks. In *IEEE Symposium on Security and Privacy*, 2020.
- [62] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In *International Symposium on Microarchitecture*, 2020.
- [63] Zhi Zhang, Jiahao Qi, Yueqiang Cheng, Shijie Jiang, Yiyang Lin, Yansong Gao, Surya Nepal, Yi Zou, Jiliang Zhang, and Yang Xiang. A retrospective and futurespective of rowhammer attacks and defenses on dram. *arXiv preprint arXiv:2201.02986*, 2022.

A SPECint 2006

SPECint 2006 is an industry standard benchmark suite intended for measuring the performance of CPU and memory. For this suite, we launch 12 integer programs with a specific configuration file (i.e., *linux64-amd64-gcc43+.cfg*) and summarize the benchmark results in Table 5. As we can see from the table, the overhead of $\Delta_{\pm 6}$ (i.e., 0.75%) is a bit higher than that of $\Delta_{\pm 1}$ (i.e., 0.04%) although the row distance of $\Delta_{\pm 6}$ is an order of magnitude larger than that of $\Delta_{\pm 1}$.

Programs	SoftTRR Overhead	
	$\Delta_{\pm 1}$	$\Delta_{\pm 6}$ (default)
perlbench	0.47%	1.42%
bzip2	-0.61%	1.52%
gcc	0.00%	0.51%
mcf	-2.08%	-2.08%
gobmk	0.30%	0.60%
hmmer	0.41%	0.83%
sjeng	0.00%	0.26%
libquantum	0.00%	0.59%
h264ref	0.00%	0.89%
omnetpp	0.32%	0.00%
astar	0.97%	2.60%
xalancbmk	0.63%	1.89%
Mean	0.04%	0.75%

Table 5: SPECint 2006 benchmark results.

B Artifact

Abstract

This artifact contains a prototype implementation of SoftTRR that protects level-1 page tables from rowhammer attacks. It works as a loadable Linux kernel module without any modifications to the kernel.

Scope

This artifact is effective and practical in protecting level-1 page tables from being corrupted by rowhammer attacks. Particularly, it leverages realistic and reasonable software refreshes to mitigate a rowhammer attack where a hammered row can be up to 6-row away from a targeted row hosting level-1 page tables.

Contents

This artifact has 6 main source files, which are briefly introduced as follows. *defense.c* is to initialize our kernel module such as registering dynamic hooks to relevant kernel functions (we rely on a third-party inline hook library

https://github.com/cppcoffee/inl_hook that resides in the directory of *inl_hook* in our repository). *victim_handler.c* collects level-1 page-table pages and their physically adjacent pages. *pgfault.c* monitors `do_page_fault` to trace memory accesses to pages that are physically adjacent to level-1 page-table pages. *rbtree.c*, *dramaddr.c* and *kernel_symbol.c* provide some helper functions such as maintaining important data structures and DRAM address mappings, parsing relevant kernel symbols, etc. We refer the readers to our repository for more details.

Hosting

This artifact is available at <https://doi.org/10.6084/m9.figshare.19721692>.

Hardening Hypervisors with Ombro

Ethan Johnson, Colin Pronovost, and John Criswell
Department of Computer Science, University of Rochester

Abstract

This paper presents Ombro, a low-level virtual instruction set architecture (vISA) which enforces compiler-based security policies on real-world commodity hypervisors. We extend the Secure Virtual Architecture (which itself extends the LLVM compiler’s Intermediate Representation) to support the full set of hardware operations needed to run an x86 commodity hypervisor used in some of the world’s largest public clouds, namely, the Xen 4.12 hypervisor, running in full hardware-accelerated mode using Intel’s Virtual Machine Extensions (VMX). We have ported Xen 4.12 to the Ombro vISA and demonstrated that it can run unmodified guest VMs of real-world relevance (namely, Linux guests under Xen’s HVM and PVH modes). Furthermore, to demonstrate Ombro’s ability to harden hypervisors from attack, Ombro implements control flow integrity and the first protected shadow (split) stack for x86 hypervisors. Our performance results show that Ombro achieves this protection without imposing measurable overheads on most application benchmarks.

1 Introduction

Various ideas have been proposed and demonstrated that can improve hypervisor security against low-level attacks such as memory safety vulnerabilities. One such approach would be to re-implement the hypervisor in a safe language such as Rust [1], but this is considered prohibitively labor-intensive for real-world hypervisors such as Xen [3], VirtualBox [46], or Hyper-V [42]. Another is to provide a whole-VM trusted execution environment (TEE) that protects VMs from a compromised hypervisor by isolating and removing most of the hypervisor from the trusted computing base [32, 41, 55, 56]; this approach is powerful and promising but tends to impose heavyweight requirements (e.g. moving the non-trusted portion of the hypervisor into VMX/SVM’s non-root (guest) mode, which imposes high overheads e.g. on VM exit handling) [41, 56] or substantial modifications to hardware [32, 55]. Enclave-based TEEs such as Intel’s SGX [30] offer similar benefits but provide a more functionally limited operating environment compared to whole-VM TEEs. Designs such as HyperSafe [51], which adds lightweight control flow integrity (CFI) protection to the hypervisor, make hypervisors more resilient against attack but face an “arms race” of rapidly evolving attacks [7, 8, 27, 29], necessitating the addition of further defenses such as a shadow stack (HyperSafe [51] is vulnerable to attacks that corrupt return addresses) to remain viable.

When applied to kernel-mode software like hypervisors, hardening approaches such as CFI must account for the fact

that the raw hardware/software interface provided by the native ISA is much “messier” than the execution environment user-mode applications can expect. Low-level operations which are typically thought of as transparent and orthogonal, such as context switches, VMX/SVM guest entry/exit, page table updates, and control register modifications, are fully exposed to host-kernel-mode software. These operations present numerous opportunities for security invariants such as CFI enforcement to be undermined when software logic has been corrupted by a memory safety exploit.

Prior work on the Secure Virtual Architecture (SVA) [15, 17] addresses this issue by extending the LLVM compiler’s Intermediate Representation (IR) with *virtual instructions* (a.k.a. *intrinsic*s) that encapsulate these low-level hardware/software interactions with principled, higher-level abstractions intended for use by kernel-mode code. Program state discontinuities that could break security enforcement are prevented, since operations such as context switches and paging updates are handled safely by a thin, trusted layer of code provided by the compiler to implement the virtual instructions, whose behavior cannot be compromised by bugs in a kernel or hypervisor built upon them. To date, SVA has been used to successfully enforce security policies such as memory safety [16] and CFI [13] on commodity Linux and FreeBSD OS kernels. Initial support for hardware-accelerated virtual machines via Intel VMX has been added to SVA [34], but it lacks several key features needed to support real production hypervisors such as Xen [3], VirtualBox [46] and Hyper-V [42], and also lacks SMP (multiprocessor) support, a necessity in the modern cloud.

In this paper, we present *Ombro*, a low-level virtual instruction set architecture (vISA) designed to support the efficient and complete implementation of compiler-based security mitigations in real-world commodity hypervisors. We extend SVA to support the full set of hardware features needed to support the Xen 4.12 hypervisor [3], including virtual APIC support, model-specific register (MSR) virtualization, and I/O port virtualization. We also add the first SMP support to SVA (benefiting non-hypervisor designs as well that are based on it) and make several key design improvements to SVA’s existing VMX support [34] to address shortcomings in performance and its ability to integrate with the Xen codebase. We have ported Xen 4.12 to the Ombro vISA and demonstrate that it can run unmodified guest VMs of real-world relevance (Linux guests under Xen’s HVM and PVH modes) with negligible performance impacts on most application benchmarks. Additionally, as a case study in the kinds of hypervisor security mitigations whose design and implementation Ombro sup-

ports, we demonstrate the implementation of control flow integrity with return address protection (shadow/split stack) using the tools provided by Ombro, and that these mitigations add no further performance impacts to guest operations.

To summarize, our contributions are as follows:

- We have enhanced the SVA vISA developed in Shade [34] to support a full-featured production-quality hypervisor (namely, Xen). We have also added symmetric multiprocessing (SMP) support to the SVA vISA.
- We have developed techniques that ensure that bugs in hypervisors cannot break return address and control flow integrity. We have created a prototype of a system, dubbed Ombro, that uses our enhanced SVA vISA to enforce these policies on a production-quality hypervisor.
- We have ported the Xen hypervisor to Ombro. This is the first full port of a full-featured production-quality hypervisor to the SVA virtual instruction set.
- We have evaluated the performance of Ombro and found that the vISA imposes negligible performance impacts on most guest application benchmarks. We have also found that the addition of CFI and return address protection imposes no measurable overheads.

2 Background

Ombro employs *virtual instruction set computing* (VISC) [5, 16] to ensure that its security guarantees (which mitigate control-flow hijacking) are not bypassed via low-level interactions between the hypervisor and the x86 hardware. Here, we present background information on VISC, the VISC-based Secure Virtual Architecture (SVA), and its features and limitations relevant to Ombro.

2.1 Virtual Instruction Set Computing

Virtual instruction set computing (VISC) [5] is a system design in which the instruction set to which software is compiled (the *virtual instruction set* or *vISA*) is decoupled from the instruction set implemented by the processor (the *native instruction set*). A trusted code generator translates code from the virtual instruction set to the native instruction set. This translation can occur at any time (at compile time, link time, install time, boot time, or just-in-time during program execution). The defining characteristic of VISC is that all software in the system must be translated from virtual instruction set code to native instruction set code. In an idealized theoretical VISC system, this includes applications as well as system software (e.g. operating system kernels and hypervisor executives), i.e., all code on the system must target the virtual ISA rather than the native ISA. Practical designs may elect to relax this requirement within specific domains (e.g. applications or guest VMs running in less-privileged hardware modes) to maintain support for existing native-code applications.

Secure Virtual Architecture (SVA) [15, 16] is a VISC infrastructure that leverages the trusted code generator to enforce security policies on all software in the system stack, including the OS kernel and (optionally) library and application code. Because software must be translated by SVA’s trusted code generator, SVA can instrument code during native code generation to enforce security policies. SVA’s virtual instruction set is an extended version of the original LLVM Intermediate Representation (LLVM IR) [39], allowing SVA to use aggressive static analysis to optimize away provably unnecessary run-time security checks.

SVA extends LLVM IR with new virtual instructions (called *intrinsic*s) to support low-level privileged operations such as I/O, MMU configuration, and context switching in kernel-mode software without the need for native assembly code or direct access to privileged in-memory hardware data structures (page tables, etc.) [15]. These *intrinsic*s are implemented by a small library of trusted code (the “SVA-OS runtime”) which is, architecturally, considered part of the compiler, and linked or inlined into the target program as necessary during translation from virtual to native code. The resultant vISA provided to kernel programmers is designed to make it impossible to express computations that would violate the security policies specified for the system. To the extent that goal cannot be ensured statically, the SVA-OS implementation vets inputs and sanitizes outputs at runtime to prevent raw hardware functions from being used in unsafe ways.

To prevent attacker-compromised kernel-mode software from simply bypassing the vISA by executing native code, newer versions of SVA [13–15, 22, 34] enforce code segment integrity on the kernel by using software fault isolation [50] to prevent it from utilizing any kernelspace page-table mappings that are both writable and executable. This ensures that all native code has been translated or validated by the SVA code generator (either ahead of time or by request at runtime) and contains any necessary instrumentation while not containing privileged native instructions that would bypass the vISA.

Because it is possible to fully implement kernel-mode system software (e.g., OS kernels or hypervisors) using the SVA vISA’s virtual instructions, compiler-based security transformations such as control flow integrity (CFI) [4], software fault isolation (SFI) [50], or memory safety [20] enforcement can be performed on it at the LLVM IR level without “blind spots” arising from opaque native assembly or instructions with privileged side effects. SVA has been used to safely and efficiently support a variety of security policies and popular OS kernels over the years. The original SVA prototype enforced both spatial and temporal memory safety on the Linux 2.4 kernel [15, 16]. Subsequent iterations exchanged full memory safety for low-overhead CFI enforced on the FreeBSD 9.0 kernel [13] and explored a novel application of lightweight SFI instrumentation on the kernel to protect userspace applications from a compromised kernel [14]. Other SVA derivatives have explored adding protection against cache and speculative

side-channel threats against protected userspace applications in a Virtual Ghost system [21, 22, 34].

2.2 Kernel-Mode Memory Protection in SVA

Starting with the *Virtual Ghost* project [14], SVA has supported protecting designated sections of the host-virtual address space against tampering by kernel-mode (Ring 0) code using software fault isolation (SFI) [50]. This protection can be used as a foundation for multiple security policies and can optionally protect portions of userspace as well as kernelspace (lower and upper halves of the address space).

The SVA native code translator instruments all load and store instructions within host-kernel-mode code with SFI checks that determine whether the access references a virtual address within the protected region. If so, the check detects a violation and generates a trap, allowing the SVA VM to take corrective action, such as alerting the system administrator or terminating system execution. Otherwise, the load or store is allowed to proceed normally [14].

SVA's SFI checks can be implemented using traditional bitmasking instructions [50] or a fast scheme developed for Apparition [22] based on Intel's Memory Protection Extensions (MPX) [30]. The architecture is flexible and can be readily adapted to other hardware protection mechanisms such as segmentation [30] or memory protection keys [28, 30].

The memory region(s) protected by these SFI checks can be used to store user or kernel secrets where they cannot be seen or modified by the kernel/hypervisor. Depending on the needs of particular threat models, enforcement designs, and system performance, SFI checks can be omitted on loads so as to protect data against tampering even if it does not need to be secret [14].

SVA uses this memory protection in its design to make its other enforcement mechanisms complete while maintaining high performance. For instance, SVA maintains its own direct map (one-to-one mapping of all physical memory) within the kernelspace SFI-protected region, allowing intrinsics to write to page tables and code pages while leaving the kernel's mappings to them read-only; this avoids the need to expensively switch page tables or mappings on every such intrinsic call [22]. SVA likewise uses its SFI to ensure the integrity of its own metadata, such as tables tracking the permitted and current usage types of each physical memory frame [15, 22].

In Virtual Ghost [14], Apparition [22], and Shade [34], a userspace SFI-protected region is used to hide application secrets from a compromised OS kernel. In Ombro, we will use SVA's SFI to protect hypervisor control stacks and enforce return address integrity (Section 7).

2.3 VMX Support in SVA

Shade [34] added initial support for Intel VMX to SVA. *Shade* extended the SVA *vISA* with intrinsics and conceptual idioms for management of hardware-accelerated guest VMs—specifically, VM entry and exit (world switches), extended

paging, and VMCS management—while ensuring that these newly introduced capabilities would not compromise SVA's ability to enforce the security policies introduced in prior work (specifically Virtual Ghost [14] and Apparition [22]).

VMX facilitates *world switches* (the host/guest context switches associated with VM entry and exit) by giving the hypervisor open-ended control over each state element—instruction pointer, control and segment registers, etc.—as an individual field within a special in-memory data structure called the Virtual Machine Control Structure (VMCS) [30]. Different state elements are handled differently according to their importance in maintaining consistent system operation, and this behavior can (to an extent) be controlled by the hypervisor via flags in the VMCS. Some fields can be bidirectionally saved/restored by the processor as part of the host/guest-mode transition; for others, the hypervisor is expected to load an arbitrary value into the VMCS to which the register will be reset on VM exit. Others, such as the general purpose registers, are untouched by entry and exit, requiring the hypervisor to save and restore them itself.

This makes security enforcement on hypervisors challenging because the architecture implicitly assumes the hypervisor can be trusted. The ability to set host-state fields and entry/exit control flags in the VMCS affords countless opportunities for a compromised hypervisor to exploit VM exit to escape CFI enforcement and other security measures. Besides the instruction pointer itself, fields such as the stack pointer, segment registers, and control registers (which can disable security features such as protected mode, No-Execute (NX) pages, and SMEP [30]) can completely redefine the hypervisor's environment, rendering many protections useless.

Shade addresses these issues in SVA by encapsulating the VM entry/exit process (*VMLAUNCH/VMRESUME*) into an SVA intrinsic, *runvm*, which handles switching of sensitive state during world switches [34]. *runvm* has the semantics of a self-contained function call, contrasting with the broad, open-ended modification of system state possible with the native interface. *Shade* keeps context-switched guest state for the host and each guest VM within SFI-protected SVA internal memory (Section 2.2), providing access to individual guest state fields only through targeted intrinsics. The VMCS itself is likewise stored in protected memory and accessible to the hypervisor through intrinsics (*read/writenvmcs*) which only permit access to non-sensitive fields (or *vet/sanitize* input/output for partially sensitive fields). VMCS fields related to features that SVA needs to control at a higher level (e.g. extended paging) are blocked by *read/writenvmcs*, forcing the hypervisor to use the appropriate higher-level intrinsics.

While *Shade* laid important high-level groundwork for safely supporting the use of VMX acceleration in an SVA-based system, it fell short of being able to support a full-scale commodity hypervisor like Xen (or even a lightweight virtualization support driver like KVM [36]). *Shade* was developed and evaluated with a minimalist “toy” hypervisor that exer-

cised core VMX operations without the complexity of a full hypervisor. This facilitated the initial design and debugging of complex intrinsics such as `runvm`, but precluded an end-to-end performance evaluation and left unclear the question of whether the design choices made in the vISA would truly be conducive to porting a real hypervisor without invasive code changes or performance impacts. Shade also lacked support for key VMX features important to real-world hypervisors such as accelerated interrupt controller (APIC), model-specific register (MSR), and I/O port virtualization [30], and only supported vetting of a small subset of VMCS fields. Our work remedies these shortcomings, allowing us to port Xen 4.12 to the SVA vISA and use it to enforce and evaluate a security policy of real-world interest (return address protection for CFI) on a real-world hypervisor.

3 Threat Model

Our threat model assumes that we have a system running a single bare-metal hypervisor, such as Xen, on the hardware. This hypervisor hosts one or more guest virtual machines running various operating systems. The hypervisor is benign but may have exploitable memory safety errors that permit control-flow hijacking attacks such as return-to-libc [49] and return-oriented programming (ROP) [47] attacks. As we want to mitigate advanced control-flow hijacking attacks [7, 8, 18, 27], our defense must protect the integrity (but not necessarily confidentiality) of return and return-from-interrupt addresses. Non-control data attacks [9], Data-Oriented Programming (DOP) attacks [29], and other memory safety attacks that do not corrupt return addresses, function pointers, and other control data are out of scope.

4 Design

In this work, we present three major design contributions:

1. We extend and improve upon the SVA virtual instruction set architecture (vISA) of the Shade [34] project to efficiently support the full set of hardware features needed to run the Xen 4.12 hypervisor [3] in support of guest VMs accelerated using the Virtual Machine Extensions (VMX) [30] features of modern Intel x86-64 processors.
2. We extend the SVA vISA to support symmetric multiprocessing (SMP), an essential feature of modern systems that historical SVA designs notably lacked.
3. We present a design for an efficient and straightforward scheme enforcing forward-edge control flow integrity (CFI) with return address integrity (i.e. backward-edge CFI). This serves as a case study in how the SVA vISA can be used to support sound and efficient enforcement of security policies on commodity hypervisors, and also represents an advancement in its own right on the state of the art [51] as the first efficient protected shadow stack design for a hypervisor.

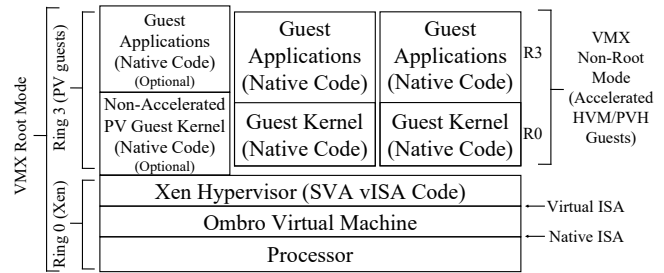


Figure 1: Ombro Architecture

Figure 1 depicts the overall architecture of Ombro as used to enforce security policies on the Xen 4.12 hypervisor.¹ Ombro permits the hypervisor to run in the processor’s highest-privileged mode—Ring 0 of VMX root mode—without relying on hardware privilege isolation to enforce security policies on the hypervisor. The hypervisor is compiled to the SVA vISA rather than native code, preventing computations from being expressed that could violate security policy. The *Ombro Virtual Machine’s*² SVA-OS runtime support library—a thin layer of trusted native code provided by the SVA native code translator (i.e. the compiler; see Section 2.1) to implement the dynamic security checks required to safely implement the vISA on native hardware—runs alongside the hypervisor in this fully privileged mode, having been linked or inlined directly into the compiled hypervisor by the SVA native code translator. Software fault isolation (SFI) [50] (Section 2.2) and control flow integrity (CFI) [4] provided by SVA ensure that the hypervisor, despite running with full hardware privileges, cannot compromise the Ombro Virtual Machine or escape the vISA to run unrestricted native code.

Notably, Ombro does *not* require guest VM code (or host userspace application code for an OS-resident hypervisor) to be compiled to the SVA vISA. The hypervisor/host OS continues to employ standard x86 privilege isolation features (namely, privilege rings and VMX root/non-root modes) to isolate guest VMs and userspace applications from the hypervisor/kernel and each other, allowing guests and applications to run unmodified native code. Only the hypervisor/host kernel itself must be ported to the SVA vISA so that it can be safely controlled within the processor’s most privileged mode.

Section 5 presents the extensions and improvements we make to the Shade [34] version of the SVA vISA to support a full commodity hypervisor (Xen 4.12). Section 6 describes specific enhancements to the SVA design that are necessary to support multi-processor systems, addressing a weakness in prior SVA work. Section 7 presents a design that utilizes the SVA vISA to efficiently and soundly enforce return address

¹Our design supports any x86-64 hypervisor in principle, including those integrated with a host OS kernel; for concreteness, we focus on Xen in this work. In an OS-resident hypervisor, host userspace applications take the place of non-accelerated paravirtual guests in Ring 3 of VMX root mode.

²“Virtual machine” here refers to the language-theoretic sense of the term, not to the concept of a “guest virtual machine” provided by hypervisors. We aim to consistently use “guest VM” to refer to the latter throughout this work.

integrity on Xen to defend the hypervisor against advanced control-flow hijacking attacks.

Guest Virtualization Modes Supported. The Xen 4.12 hypervisor, for legacy reasons, supports a variety of operating modes for guest VMs on the x86-64 platform, which can be used simultaneously for different guests [54]. These represent a continuum of hardware acceleration usage and guest OS support, ranging from the traditional non-accelerated paravirtualization described in the original Xen paper [23] (referred to as “PV” mode in Xen’s documentation [54]), to VMX-accelerated full virtualization supporting native guests (“HVM” mode), to a modern paravirtual approach utilizing VMX acceleration with Xen-aware guests (“PVH” mode).

Because classic PV mode is being de-emphasized by the Xen project and likely to be phased out in the future [54], Ombro’s design only supports VMX-accelerated guests (i.e. HVM and PVH) and does not provide a complete set of intrinsics for the hypervisor to support PV guests running in usermode (host Ring 3). However, doing so would be a straightforward extension of the existing SVA vISA, which fully supports [14–16, 22, 34]) OS kernels with userspace applications in Ring 3. Our prototype (Section 8) does exactly this (in a limited way with some shortcuts) for implementation convenience. At times, we refer to PV concepts in design sections for the sake of clarity to readers familiar with Xen. They are not, however, relevant to our security design, as the design is intended for a modern Xen installation utilizing HVM/PVH for all guests including the control domain (*dom0*).

Terminology: Guest VMs, Domains, vCPUs, etc. SVA interfaces (following Intel’s convention) use the term “guest virtual machine” to refer to a singular guest CPU virtualized by a hypervisor. Each such guest CPU exists in one-to-one correspondence with a Virtual Machine Control Structure (VMCS) [30]. From the physical CPU’s and SVA’s perspective, it does not matter how the hypervisor may choose to group those virtual CPUs together. However, this distinction is important to Xen, which refers to the overall VM (which may include multiple emulated CPUs) as a “domain” and individual virtualized CPUs as “vCPUs”. Each vCPU thus corresponds to exactly one VMCS, and to one “guest VM” from SVA’s perspective. Xen performs context switches on a per-vCPU basis, not per-domain (different vCPUs from the same or different domains are context switched as independent entities), which is in line with Intel’s and SVA’s perspective. In this paper, we sometimes refer to “vCPUs” rather than “guest VMs” when this distinction is important.

5 vISA Additions and Improvements

Our experience porting Xen to the SVA vISA developed for Shade [34] led us to extend and improve upon the vISA, adding missing support for hardware virtualization elements used by Xen and mitigating disruptions to Xen’s performance and code structure. We describe these improvements below.

5.1 Securing Higher-Level VMX Features

The Intel VMX feature set is controlled primarily through control and state fields in a large (page-sized) in-memory, per-vCPU data structure called the Virtual Machine Control Structure (VMCS) [30]. Most VMX settings and subfeatures are controlled straightforwardly via individual VMCS fields or bits within a field consolidating similar controls. Some subfeatures, however, are more complex and are spread across multiple VMCS fields; some even utilize subsidiary control pages that the hypervisor must provide and link into the main VMCS by storing (host-)physical pointers into specific fields.

Shade’s vISA support for the VMCS (Section 2.3) is insufficient to manage multi-field controls and substructures because its `read/writevmcs` intrinsics operate on individual fields and cannot readily account for behavioral dependencies between them (e.g. fields activated by bits in other fields, or structure pointers that must be set prior to enabling a feature in a different field). Thus, Shade must err on the side of caution wherever invalid field combinations could lead to security holes or undefined behavior: it categorically blocks the hypervisor from writing to such fields and forces the features they control to be disabled or utilized in a hardcoded fashion preconfigured by the SVA runtime.

While this achieved Shade’s goal of ensuring security for the host system while supporting basic VMX functionality, it locks the hypervisor out of performance- and functionality-critical VMX features such as extended paging, interrupt controller (local APIC) virtualization, and MSR³ and I/O port virtualization. Shade provided higher-level intrinsics supporting extended paging by extending SVA’s existing support for vetting host page table updates [34] but did not attempt to design vISA support for APIC, MSR, or I/O port virtualization.

Ombro extends the design of the vISA to support APIC, MSR, and I/O port virtualization as first-class idioms; Table 1 summarizes the new intrinsics. As all three of these features entail both multi-VMCS-field dependencies and substructures linked into the VMCS (and APIC virtualization interacts with MSR virtualization), we address them via similar techniques.

MSR and I/O virtualization are the more straightforward of these to support. By default, VMX guests are blocked from accessing MSRs or performing port I/O (i.e., `rd/wrmsr` and port I/O instructions cause VM exits) [30]. When MSR or I/O virtualization is enabled in the VMCS, the processor will selectively allow guests to read or write particular MSRs or I/O ports based on whether their corresponding bits are set in the *MSR* and *I/O bitmaps*, which are substructures linked via host-physical pointers in the VMCS. This hardware design has multiple security implications in an SVA system.

Firstly, as MSRs control privileged processor features (including crucial security features like long mode and page-

³*Model-specific registers* (MSRs) are a class of indexed control and information registers used extensively in the x86 ISA to manage privileged processor features [30]. They have widely varying semantic and security implications and represent a substantial portion of the x86 ISA’s complexity.

Table 1: APIC, MSR, and I/O Virtualization Intrinsics

Name (Arguments)	Description
<code>vlapic.enable</code> (<code>paddr virtual_apic_page</code> , <code>paddr apic_access_page</code>)	Enable APIC virtualization for the active VM using the xAPIC (MMIO) interface
<code>vlapic.enable_x2apic</code> (<code>paddr virtual_apic_page</code>)	Enable APIC virtualization for the active VM using the x2APIC (MSR) interface.
<code>vlapic.disable</code>	Disable the currently active VM's local APIC (or use exit-based virtualization).
<code>posted.interrupts.enable</code> (<code>u8 vector</code> , <code>paddr descriptor</code>)	Enable posted interrupt processing for the currently active VM.
<code>posted.interrupts.disable</code>	Disable posted interrupt processing for the currently active VM.
<code>msr.intercept.{get,set,clear}</code> (<code>int vmid</code> , <code>u32 msr</code> , <code>enum rw</code>)	Get, set, or clear an MSR intercept for the specified VM.
<code>io.intercept.{get,set,clear}</code> (<code>int vmid</code> , <code>u16 port</code>)	Get, set, or clear an I/O port intercept for the specified VM.

level execute permissions [30]), the vISA cannot allow untrusted host software (e.g. the hypervisor or host kernel) to access them arbitrarily. Thus, neither of the `rdmsr` or `wrmsr` instructions are present in the SVA vISA; relevant processor features are managed through higher-level vISA idioms or controlled directly by SVA. A guest VM, however, can execute native (non-vISA) kernel-mode code; its access to MSRs is therefore constrained only by the MSR bitmaps. Therefore, it is necessary for Ombro to constrain the settings of the MSR bitmaps on a per-MSR level.

Secondly, the bitmaps themselves, being VMCS substructures addressed via raw host-physical pointers, would represent a security hole if the hypervisor were allowed to control them directly. A compromised hypervisor could configure the processor to use bitmap addresses corresponding to arbitrary physical memory, including SVA-protected pages (Section 2.2), allowing it to trick SVA into overwriting protected memory (since SVA must write to the bitmaps to ensure guests exit when accessing security-sensitive MSRs) or infer the contents of protected memory based on a guest's behavior.⁴

Ombro addresses these issues by allocating and taking ownership of the MSR and I/O bitmaps itself in protected memory, as it does with the VMCS (Section 2.3). The `msr_intercept.clear` intrinsic (Table 1) checks the provided MSR index against a whitelist of known-safe MSRs that guests can access without compromising Ombro's security policies. `io_intercept.clear` does not need to impose any restrictions under Ombro's threat model, as the only need is to prevent abuse of the bitmap substructure (per-port filtering is available for potential use under other threat models).

APIC virtualization (APICv) poses similar challenges but is more complex. Modern processors support both legacy *xAPIC mode* (in which the APIC is controlled via a memory-mapped I/O (MMIO) interface) and the newer *x2APIC mode* (which is controlled via MSRs) [30]. Traditionally, hypervisors would

⁴While Ombro's threat model (Section 3) does not require confidentiality of SVA-protected memory, other SVA-based systems such as Virtual Ghost [14] and Shade [34] do, making this a relevant design consideration.

configure VMX to force a VM exit on all APIC accesses, either via extended paging (for the xAPIC MMIO interface) or by configuring the MSR bitmaps to force exits for APIC-related MSRs (for x2APIC). This allows the hypervisor to fully emulate the APIC in software, but is slow.

APICv allows certain common APIC accesses by the guest to be virtualized in hardware without a VM exit. The hypervisor provides a *virtual-APIC page* in memory whose fields stand in for the real APIC's registers when a guest attempts to access them [30]. Guest reads to APIC registers via the MMIO (xAPIC) or MSR (x2APIC) interfaces see the values provided by the hypervisor in the virtual-APIC page. Guest APIC writes are virtualized by hardware without a VM exit in situations involving the task- and processor-priority registers, end-of-interrupt signaling, and self-IPIs; unvirtualizable writes are stored to the virtual-APIC page followed by a VM exit so the hypervisor can handle them conventionally.

Relatedly, *posted-interrupt processing* [30] allows a hypervisor to send interrupts to a guest running on a different processor without forcing that guest to VM exit. When a processor in guest mode receives a (real) inter-processor interrupt to a specified *notification vector*, it will (without exiting) check an in-memory *posted-interrupt descriptor* to see if one or more virtual interrupt records have been deposited within (e.g. by the hypervisor on the sending CPU), and if so, deliver them to the running guest through its virtual APIC.

Both features effectively map pages of host-physical memory into a guest's address space with (limited) write access, posing a clear security risk under our threat model, as this could be used to defeat SVA-protected memory (e.g. the return address stacks described in Section 7). Unlike with MSR and I/O port virtualization, it is not convenient to simply have Ombro take ownership of the virtual-APIC page and posted-interrupt descriptor in protected memory, as the hypervisor needs to frequently read and write to them for normal operation. Thus, to allow the hypervisor to safely control these pages, our APICv intrinsics (Table 1) check and record the VMCS's references to them in SVA's memory metadata tables (which track the usage of every 4-kB physical memory frame; see Sections 2.2 and 6.1) as if they were page mappings accessible to the hypervisor. The hypervisor is thus only allowed to use non-sensitive pages it "owns" as APICv VMCS substructures. Additionally, Ombro ensures that the vector used for posted-interrupt notifications does not overlap with any intercepted by SVA.

5.2 Guest Context Switching Optimizations

The x86 platform supports context switching of floating-point unit (FPU) state using the `XSAVE` and `XRSTOR` instructions [30]. These instructions save or load a processor's entire FPU state, as well as that of several non-FPU features such as vector and memory protection extensions, as a several-kB monolithic data structure; system software is thus architected to perform these slow operations as infrequently as possible.

Xen refrains from using the FPU during its own execution, leaving guest state untouched and allowing Xen to only perform `XSAVE/XRSTOR` when context switching from one guest vCPU to another.

Shade [34] took the straightforward but inefficient approach of context switching FPU state on every VM entry/exit. This provided a clean abstraction wherein no guest state is ever active on the processor in host mode or vice versa, but porting Xen to the SVA vISA for Ombro showed this to be a flawed design, yielding more than 200% overhead over baseline (non-SVA) Xen in our no-op hypercall (VM entry/exit) microbenchmark and unacceptably high overheads of up to 60% on VM-exit-heavy guest workloads (Section 10).

Ombro eliminates this source of overhead by extending the SVA vISA's existing *thread* abstraction, which can be context switched independently of processor privilege level transitions, to include guest VM (VMX non-root mode) state in addition to userspace host process (Ring 3) state as in prior SVA work [14, 16]. This better matches how real-world hypervisors like Xen model context switching and allows FPU state to be switched on vCPU context switches instead of requiring it on every VM entry/exit. Because slow-switching elements of the guest's state (the FPU, some MSRs, etc.) are thus active even while the system is in host mode, this provides a slightly less flexible programming model to the hypervisor than in Shade, but these limitations are non-issues in practice and can be worked around: Xen already avoids using the FPU in hypervisor context; an OS-integrated hypervisor would simply allocate separate SVA threads for guest VMs and host processes; and a hypervisor could free up the FPU for itself by context switching to a dummy SVA thread not associated with a guest VM or host process.

5.3 VMCS Management Optimizations

Ombro makes two additional improvements to the SVA vISA to eliminate overheads related to VMCS management induced in Xen by the Shade [34] design. The foremost of these is that, unlike the native ISA, Shade only permits a single VMCS to be loaded on a CPU at a time. The native ISA only permits one VMCS to be *active* at a time, but others need not be unloaded to load a new one [30], allowing them to remain cached by the hardware for future context switches. This distinction is crucial to performance; we found via an informal benchmark that modifying Xen to explicitly clear (flush) the outgoing VMCS in every context switch induces unacceptably high (over 4x) runtime overhead on guests when the machine's physical CPUs are oversubscribed (i.e., when vCPU context switches are frequent). Ombro addresses this limitation by loosening the vISA to allow multiple loaded VMCSes to coexist, relying on the mutex in the SVA thread structure (Sections 5.2 and 6.1) to ensure a VMCS cannot be loaded on multiple CPUs at once (which is undefined behavior in the native ISA [30]).

The second improvement changes VMCS initialization (the `allocvm` intrinsic) to provide benign defaults for all security-sensitive VMCS and guest state fields rather than requiring the hypervisor to specify them up-front. While VMCS construction is not performance-critical for Xen, it occurs early in vCPU creation before Xen has determined most of the guest's initial state, making it awkward to port Xen to use Shade's interface. Ombro's interface is more general and agnostic to hypervisor design choices.

6 SMP Support in SVA

Ombro adds symmetric multiprocessing (SMP) support to SVA that all previous SVA systems [13–16, 21, 22, 34] lacked. This required several changes to the internal design of the SVA runtime library to make it thread-safe and to add support for multi-processor TLB coherency. However, we made *no* changes to the outward-facing SVA-OS virtual instruction set; the original design [15–17] proved general enough to be applicable to both uni- and multi-processor systems.

6.1 Thread Safety and Reference Counting

SVA maintains several data structures in its internal protected memory (Section 2.2) to track system state that it maintains on behalf of the hypervisor/OS kernel and to ensure that its intrinsics are not used to configure the system in a way that could undermine other security protections. These include thread structures used for context switching host processes [16] and guest vCPUs (Section 5.2) and a table tracking typed references to each physical memory frame to prevent host-kernel-mode software from using its control over the MMU to evade SVA's memory protections or code integrity enforcement [15].

As these structures must be thread-safe in a multi-processor system, Ombro adds locking to SVA's thread structures and to each entry in the frame usage table. Intrinsic calls attempting to load or save a thread or to change a frame's usage type must obtain the relevant lock to prevent races. Incrementing/decrementing a frame's reference count in the table when a page mapping is updated is a lock-free operation utilizing an atomic compare-exchange loop to perform the update while checking for integer overflow/underflow.

Additionally, Ombro expands the frame reference counts themselves to separately count read-only and writable page mappings to each frame. Prior SVA work [15, 22, 34] did not allow system software to create *any* mappings to SVA-protected frames even when they only require tamper-protection and not confidentiality (e.g. kernel code or page tables). This required ad-hoc (and OS-specific) handling in SVA of special cases like the kernel's direct map and read access to page tables. Ombro allows these to be handled through ordinary intrinsic calls, making SVA more system-agnostic and allowing Xen to continue supporting non-VMX PV guests under Ombro.

6.2 TLB Shootdowns

In an SVA system, physical memory frames used for security-sensitive purposes such as page-table pages, host-kernel-mode code, or SFI-protected memory (e.g. return address stacks in Ombro—see Section 7) are tracked by SVA so it can prevent system software from mapping them into the virtual address space with inappropriate permissions or outside the SFI-protected region [14, 15, 22]. To prevent use-after-free attacks based on stale TLB entries, SVA flushes the TLB whenever a frame’s usage type changes and one or both of the types involved are security-sensitive (because x86 does not support selective TLB flushes based on physical addresses [30], a full TLB flush must be used).

On multiprocessor systems, this TLB flush must include *all* processors, necessitating *TLB shootdowns*. Ombro implements this by broadcasting an inter-processor interrupt (IPI) [30] to all processors at a reserved interrupt vector, which is received by an SVA handler that performs each local TLB flush. The processor initiating the shutdown will not release its lock on the frame’s usage type until all other processors have acknowledged completion of the flush, ensuring that software cannot create a conflicting mapping based on the new type that would violate security policy.

7 Return Address Integrity

To defend against advanced control-flow hijacking attacks as described in our threat model (Section 3), Ombro must protect the integrity of return and return-from-interrupt addresses in Xen. We address this by using the vISA primitives provided by SVA to implement a *split stack* in Xen, where return addresses are stored on one stack (called the *control stack*) while local variables are stored on a separate stack (called the *data stack*). The control stack is protected against tampering using SVA’s kernel-mode memory protection mechanism (Section 2.2), while memory writes utilizing dynamic pointers or offsets that could be controlled by an attacker are only permitted to access the data stack in ordinary (unprotected) Xen memory.

7.1 Security Guarantees

Ombro ensures return address integrity (all functions return control flow to their dynamic callers) by enforcing the following invariants on the hypervisor at runtime:

Invariant 1. *Function calls always save the return address on the control stack, or do not save any return address (e.g. tail calls).*

Invariant 2. *Returns will always retrieve the return address from the correct location on the control stack, i.e. into which the return address was saved by the matching dynamic caller.*

Invariant 3. *Control stacks cannot be corrupted by any code outside of trusted SVA-OS intrinsics, even when memory safety errors are exploited.*

Invariant 4. *System software cannot use an SVA-OS intrinsic to tamper with a control stack’s contents or a control stack pointer on its behalf.*

7.2 Enforcement Design

In a split stack design, the control and data stack are tracked by separate stack pointers that can be incremented and decremented independently [10, 59]. In Ombro, we use the native x86 stack pointer register, `RSP`, for the control stack, so that call and return instructions naturally use the control stack for return addresses. This maintains Invariant 1 and contributes to Invariant 2. The data stack is tracked by a free general-purpose register reserved from the callee-saved set (`R15` in our prototype). The compiler is modified accordingly to facilitate this; function prologues and epilogues create and destroy stack frames using the data stack pointer, leaving the control stack pointer to be adjusted only by the return-address pushes and pops performed by call and return instructions.

Call and return instructions are the only ones permitted to modify the control stack pointer `RSP` outside of SVA-OS intrinsics. They always respectively decrement or increment `RSP` by exactly 8 bytes (the Ombro compiler will not emit returns that pop additional values off the stack), ensuring that only the relevant return address is affected. No other data is stored on the control stack besides the single return address pushed by each call and popped by the corresponding return; the data stack is used for local variables, argument passing, and callee-saved registers. As forward-edge CFI prevents functions being entered except through a call (non-tail-call jumps can only target another location within the current function), calls and returns are guaranteed to occur in correctly nested (matching) order. Hence Invariant 2 is ensured.

Because calls and returns occur in nested order, underflow of the control stack pointer cannot occur. Overflow is addressed by placing a guard page at the end of each control stack; guard pages are marked as invalid in the page tables, ensuring that any attempt to read or write beyond the space allocated for a control stack will be intercepted and prevented by an SVA fault handler.

Ombro instruments all host-Ring-0 code outside of SVA-OS intrinsic implementations with SFI checks on memory stores (Section 2.2), ensuring that any attempts to write to a protected virtual address region will be caught and prevented. SVA’s enforcement of code segment integrity (Section 2.1) in conjunction with forward-edge CFI [13, 22] ensures that these SFI checks cannot be bypassed even in the presence of memory safety errors. The control stack is allocated within the SFI-protected virtual address region by SVA on Xen’s behalf. Only call and return instructions are exempted from SFI checks (so that they can access the control stack as intended); these are generated by the compiler such that they always access the stack using a predictable static offset from `RSP`, so they cannot be used to access any other location in service of an exploit. Hence Invariant 3 is ensured.

The SVA vISA provides no means for system software to set or adjust the host-Ring-0 control stack pointer `RSP`, or to write to any location on a control stack, except pushing/popping from it via calls and returns. SVA allocates a control stack for each CPU (Xen allocates hypervisor stacks on a per-physical-CPU basis) and points `RSP` to it as a side effect of SVA’s boot-time per-CPU initialization; thereafter, SVA maintains the integrity of `RSP` across all intrinsic calls, context switches, and VM entry/exit. Per our threat model (Section 3), the hypervisor is considered benign prior to exploitation by a memory safety error; SVA initialization occurs during early boot before significant attack surfaces become available (there is no network yet, nor have any guest VMs been started, including the `dom0` control domain). The SVA-OS intrinsic implementations themselves are part of the trusted computing base and thus assumed to correctly implement the vISA, ensuring that Invariant 4 is upheld after initialization.

8 Implementation

The prototype we built to evaluate Ombro is based on source code for the SVA-OS runtime support library (see Section 2.1, Figure 1, and Section 4) inherited from the Shade [34] project and other previous SVA work [14–16, 22]. We significantly modernized and refactored the codebase to address limitations of prior work, which included adding multi-processor support and overhauling the page reference tracking system to be more flexible (Section 6). Overall, we improved the code to be substantially less fragile and more maintainable, and generalized aspects of the code that were specific to using SVA with OS kernels (and FreeBSD in particular), such that it could support a bare-metal hypervisor like Xen while retaining support for OS kernels (including those with integrated hypervisors). We upgraded SVA to be based on the LLVM 10.0.0 compiler [2] instead of LLVM 3.1 as in Shade [34] and prior work. In parallel, we ported the x86-64 implementation of Xen 4.12 [3] to target Ombro’s version of the SVA vISA, linking it at compile time with the SVA-OS runtime support library (as in prior SVA work).

Our software trusted computing base (TCB) relative to our threat model (Section 3) consists of the SVA runtime library (11,453 lines of code (LOC)), our CFI and SFI passes added to the LLVM compiler (792 LOC), and non-pass modifications to LLVM implementing our split stack transformation (497 LOC added/changed), totaling 12,742 LOC.⁵

Porting Xen 4.12 to the SVA vISA and supporting Ombro’s split stack transformation entailed adding/changing $\leq 3,389$ LOC⁶ (out of 313,377 total in Xen—about 1%), mostly in low-level code dealing with page tables, VM entry/exit, VMCSes,

⁵We counted non-comment/whitespace lines using `sloccount` [52] for the SVA library and passes and manually from `git diff`s for the rest.

⁶Most changes are gated behind `#ifdefs`, leaving the original code in place; we took the difference between vanilla Xen and the Ombro port using `sloccount` and added twice the number of removed lines from `git diff` as an upper bound on the undercount.

and system boot. By comparison, prior work’s port of Linux 2.4.22 to SVA [16] modified 5,066 LOC out of 632,469 total (0.8%). This shows that the difficulty of an SVA port scales roughly but not exactly with the scope of the system.

As Section 4 describes, Ombro’s security design assumes that all domains are used only in VMX-accelerated modes (HVM and PVH), but our prototype retains partial support for classic PV mode. Although Xen 4.12 supports a PVH `dom0`, we found it useful to use a PV `dom0` in order to have a working, debuggable environment while porting VMX-related components to SVA, as the `dom0` is responsible for controlling Xen and providing hardware drivers. We did not attempt to fully port Xen’s PV code to SVA, leaving some native assembly and unsafe (to our threat model, not Xen’s) workarounds in place, though this could be completed with a little extra work and minor enhancements to the SVA vISA. Since we benchmarked (Section 10) within a PVH `domU`, we do not expect significant performance differences between a PV and PVH `dom0`, especially since we used the same configuration for both Ombro and baseline Xen.

Further Implementation Experience Discussion. Some readers may be interested in further discussion of our experience porting Xen to the SVA vISA and how that experience, as well as our incremental performance evaluations throughout, fed back into our design process and motivated specific design changes. For reasons of space, that discussion is deferred to Appendix B.

9 Security Analysis

Prior work [7, 8, 18, 25, 27] has shown that defenses relying on CFI [4] to mitigate memory safety attacks must *prevent* the corruption of return addresses in order to repel advanced control-flow hijacking attacks, as static determination of the call sites to which control flow may return leaves sufficient loopholes for an attacker to perform arbitrary computation [8]. To this end, Ombro implements a split control/data stack in Xen (Section 7) and uses SVA’s SFI-based kernel-mode memory protection facility (Section 2.2) to protect the control stack (and hence return addresses) from tampering.

Zhou et al. [58] proposed a framework for evaluating the attack surface exposed by a security policy providing return-address protection, drawing from Göktaş et al.’s [27] taxonomy classifying the different types of control-flow “gadgets” that can be used by an attacker to assemble a code-reuse attack in the presence of CFI. *Call-site* (CS) gadgets begin at a return point following a call instruction, and *Entry-point* (EP) gadgets begin at the entry point to a function. Orthogonally, they classify the methods by which gadgets can be linked together to form a “chain” useful for computation: by corrupting return addresses on the stack (*return-oriented programming*), indirect jump targets (*jump-oriented programming*), or function pointers (*call-oriented programming*). Depending on the CFI policy in effect and the availability of gadgets in the target

program, it may be possible and/or necessary to mix different gadget types and chaining methods to achieve a practical chain. Pure call-oriented programming, in which only function pointers are manipulated to link gadgets, turns out to be difficult or impractical to achieve in many cases, particularly in scenarios where it is not possible for the attacker to repeatedly exploit memory safety errors; return- or jump-oriented gadget linking must typically be used to perform initial setup before the attacker can pivot to a call-oriented chain [7, 27].

Because Ombro’s split stack design prevents any corruption of Xen’s control stack (which includes return as well as return-from-interrupt addresses), return-oriented gadget linking is categorically precluded. Additionally, the SVA vISA does not include LLVM’s indirect jump (`indirectbr`) instruction and does not need it to support Xen, the Linux kernel [16], or the FreeBSD kernel [13, 14]. Thus, jump-oriented gadget linking is precluded. Cumulatively, this severely limits the ability of an attacker to assemble a useful gadget chain, particularly in single-exploit scenarios, as only call-oriented chaining is possible. That, in turn, is further limited by Ombro’s forward-edge CFI protection.

Ombro’s label-based forward-edge CFI scheme is based on that of KCoFI [13]; it is relatively coarse-grained, allowing indirect calls to target any valid function entry point (but nowhere else). Since the protected control stack already prevents return-oriented gadget chaining, and the prohibition of `indirectbr` prevents jump-oriented chaining, the net effect of adding this forward-edge CFI enforcement is to limit an attacker to a restricted form of pure call-oriented programming: only entry-point (EP) gadgets can be used and they can only be chained together via corrupted function pointers.

Ombro may be susceptible to the pure-call-oriented “Control Jujutsu” attack described by Evans et al. [25], who showed that, in popular programs, common function pointer coding patterns make it possible to assemble purely call-oriented chains of EP gadgets even when return addresses are fully protected and strong static analysis is used to limit forward-edge control flow transfers to those intended by normal program operation. Specifically, they noticed that this was possible in Apache and Nginx due to extensive use of function pointers to provide C++-style runtime polymorphism in C, which exacerbates the lack of context-sensitivity in static (label-based) CFI. We observe that Xen frequently utilizes similar code patterns, using function pointers to delegate at runtime to method implementations specific to particular virtualization modes, hardware capabilities, etc. Hence, similar attacks might be possible against Xen protected by Ombro. It is unclear, however, whether such attacks can work in practice against a bare-metal hypervisor like Xen. Evans et al. relied on the proliferation of arbitrary-code-execution system calls such as `system()` and `execve()` to invoke a shell rather than attempting to achieve arbitrary computation through code reuse alone; these elements do not exist in Xen, and similarly desirable functionality from an attacker’s perspective (e.g.,

giving the attacker’s domain *dom0* privileges or mapping a victim domain’s memory into the attacker’s) is not necessarily invoked from as many places in the codebase.

Although our prototype implementation only restricts indirect calls to calling any valid function, it would be straightforward to extend our label-based CFI approach to utilize a more precise control flow graph based on more advanced static analysis. Because Xen is a monolithic executable and does not support run-time module loading, whole-program analysis could be used to identify functions that are never address-taken (i.e. never indirectly called) and refrain from emitting CFI labels for them, eliminating them as viable targets for gadget chaining. While such functions might be reachable “downstream” of other functions that *are* called indirectly, this could make return-to-library [49] and similar short-chain code-reuse attacks more difficult, furthering the goal of making it difficult to reach functionality of ultimate interest to an attacker. (Most indirectly-called functions in Xen are focused on low-level platform-specific and scheduling-related functionality, and do not interact directly with code manipulating sensitive high-level fields such as access controls.)

Since it is built using SVA, Ombro prevents many attacks (such as code injection) even if an attacker successfully diverts control flow. Because SVA prevents tampering with host-kernel-mode code pages (i.e. Xen or SVA code) to ensure that all privileged code is compiled with the trusted vISA translator (Section 2.1), it is impossible for an attacker to pivot from code reuse to a more flexible code-injection attack, which is typically the goal of code-reuse payloads in practice [27, 35]; attackers must perform *all* malicious computation via code reuse. While ROP “compilers” do exist (e.g. [48]), they are incapable of compiling complex high-level payloads, particularly for the restrictive code-reuse modalities necessary to defeat Ombro’s CFI. This effectively limits attackers to non-control data and data-oriented programming (DOP) attacks, which are outside our threat model’s scope (Section 3). DOP could also face practical headwinds similar to Control Jujutsu.

10 Performance Evaluation

To evaluate Ombro’s performance, we selected a portfolio of real-world application macrobenchmarks (Section 10.1) that we believe are reflective of typical cloud workloads. We used the Phoronix Test Suite’s `pts/kernel` suite [40, 44] as a starting point, with the following adjustments:

- We excluded benchmarks that failed to compile or run on our test system running unmodified Xen (see system details below). We used a more recent version (2.4.48) of the Apache benchmark rather than exclude it due to its real-world importance and use by related work [41].
- Because the full `pts/kernel` suite takes days to run and includes numerous configurations of the same applications with minor differences, we ran a single configuration of

each application, selecting the longest-running one that completed in less than five minutes. Our full suite runs in approximately three hours. Phoronix runs each benchmark at least three times and until the standard deviation of all runs is less than 3.5% or a benchmark-specific cut-off threshold is met; the result reported for each benchmark is the (arithmetic) mean of these runs [45].

- We chose to add a Memcached benchmark (v1.6.9, also from Phoronix) because it is used by related work [41] and represents a stress test for Ombro’s overheads. We used the “Get” configuration with four connections.

We also developed and ran microbenchmarks (Section 10.2) for hypercall latency (VM entry/exit), EPT fault handling, and inter-processor interrupts (IPIs); our selection of microbenchmarks parallels related work [41].

For all of our experiments, we used a Dell Precision 7820 workstation with an Intel Xeon Silver 4208 (Cascade Lake) CPU (8 cores/16 threads at 2.10 GHz with 11 MB L3 cache) [11], 32 GB 2666 MHz DDR4 memory, a 256 GB M.2 NVMe SSD, and a 2 TB SATA 7200 RPM hard drive. All disk I/O for the experiments used the SSD (the hard drive was unused); the *dom0* (control VM) had direct access to the SSD and the *domU* (guest VM)’s virtual disk was directly backed by a physical partition (not a file-based disk image).

For our baseline, we ran unmodified (“vanilla”) Xen 4.12 with an Arch Linux (kernel 5.15.12-arch1-1, packages updated 2022-01-04⁷) *dom0* running in PV (traditional paravirtualization) mode. The benchmarks ran within a *domU* running the same Linux distribution in PVH (VMX-accelerated with paravirtual optimizations) mode. The *domU* was allocated 16 vCPUs and 24 GB of RAM. We used GCC 11.1.0 to compile the benchmark applications. We disabled Spectre [37] mitigations such as IBRS [30] for all benchmark runs to provide a fair comparison, since our prototype did not implement them (for reasons of time). We likewise enabled eager FPU saving in vanilla Xen as Ombro saves FPU state on every vCPU context switch (Section 5.2).

We evaluated Ombro using the implementation described in Section 8, i.e., Xen 4.12 ported to the SVA vISA and compiled with CFI, SFI, and split-stack transformations. All other configurations were the same as for the baseline.

10.1 Macrobenchmark Results

Table 2 summarizes our macrobenchmark results comparing baseline unmodified Xen (labeled “vanilla”) with Ombro. As Table 2 shows, Ombro incurs no detectable overheads (differences are within or close to noise margins) on most benchmarks, even though we selected a predominantly I/O-intensive (i.e. challenging to virtualize) benchmark set. Several exceptions, particularly Memcached, RocksDB, and LevelDB, are discussed further in Section 10.3 and Appendix A.1.

⁷Arch Linux is a “rolling” distribution without versioned releases, so the package date stands in lieu of a version number.

Table 2: Macrobenchmark Results

Benchmark	Units (↑↓ is better)	Vanilla		Ombro	
		Result	Std. Dev.	Ovhd.	Std. Dev.
PostgreSQL	TPS ↑	4266.496	26.7%	-17.84%	18.5%
PostgreSQL	ms (avg. lat.) ↓	61.644	21.0%	-16.72%	18.9%
MBW	MiB/s ↑	6694.581	0.3%	-2.88%	0.1%
BenchmarkMutex	ns ↓	39.967	0.9%	-0.67%	0.0%
PostMark	TPS ↑	4335.000	1.0%	-0.58%	1.0%
pmbench	μs ↓	0.113	4.5%	-0.55%	4.2%
OSBench (Create Files)	μs/event ↓	24.270	0.3%	-0.07%	0.8%
ctx_clock	clocks ↓	240.000	0.0%	0.00%	0.0%
OpenSSL (RSA 4096)	signs/s ↑	1516.867	0.3%	0.05%	0.2%
StressNG (RdRand)	bogo ops/s ↑	250298.447	0.0%	0.06%	0.0%
Apache	req/s	212698.913	0.4%	0.34%	0.1%
t-test1	s ↓	23.587	0.9%	0.75%	0.6%
iPerf (TCP)	Mbits/s ↑	28029.667	2.3%	0.75%	1.1%
SQLite	s ↓	83.164	0.8%	1.06%	0.7%
Hackbench	s ↓	140.056	0.8%	4.53%	1.8%
Schbench	μs ↓	18762.667	0.7%	4.59%	8.0%
IPC (TCP Socket)	messages/s ↑	473697.400	13.3%	4.63%	12.1%
LevelDB	μs/op ↓	428.367	0.1%	12.62%	0.3%
RocksDB	ops/s ↑	34557.667	0.1%	12.82%	0.3%
Memcached	ops/s ↑	46960.733	0.2%	21.81%	2.1%
Geometric mean				0.87%	

Table 3: Microbenchmark Results (TSC cycles)

Benchmark	Vanilla		Ombro		
	Cycles	Std. Dev.	Cycles	Std. Dev.	Ovhd.
No-op hypercall	750	18.5%	1465	14.2%	95%
EPT fault	4180	10.0%	5693	8.88%	36%
vCPU self-IPI	1731	12.9%	2353	14.6%	36%
IPI (vCPU→vCPU)	2966	14.6%	3531	14.4%	19%

We also ran our benchmarks with Ombro’s split stack, CFI, and SFI transforms disabled to determine if Ombro’s security instrumentation contributes significant performance overhead. Our results show no discernible difference, i.e., nearly all of Ombro’s overhead comes from the vISA itself, not the instrumentation. Appendix A.2 contains detailed results.

10.2 Microbenchmarks

Table 3 summarizes our microbenchmark results for key hypervisor operations. We ran 2²⁸ timed iterations of each benchmark (reporting the arithmetic mean) after 2¹² untimed warmup iterations. Results are in hardware timestamp counter (TSC) cycles measured using the `rdtsc` instruction (with Xen’s `rdtsc` interception disabled) before and after the operation (or sending/receiving on the respective CPUs for IPIs; our processor synchronizes the TSC across cores [30]).

Each microbenchmark includes a VM entry/exit cycle, which is minimally exercised by the no-op hypercall benchmark. Ombro’s base hypercall overhead is substantially greater than the overhead of more complex operations, which are themselves greater than our macrobenchmark overheads in Section 10.1. This indicates that VM entry/exit is the primary source of Ombro’s overhead. Since hardware acceleration is designed to make VM exits relatively rare, Ombro’s overhead only substantially impacts difficult-to-virtualize workloads that incur frequent VM exits.

10.3 Overhead Sources and Their Remedies

Prior work [41] illustrated that Memcached’s performance is highly sensitive to changes in VM entry/exit latency be-

cause it is frequently bottlenecked by the need to send inter-processor interrupts (IPIs) between guest vCPUs, e.g. by Linux’s implementation of the `futex()` system call. IPIs are efficient on a “bare-metal” (unvirtualized) system but expensive to virtualize because hypervisors cannot safely expose the interrupt controller (local APIC) to guests, requiring a VM exit to virtualize the APIC in software whenever an IPI is to be sent. Although VMX currently provides exit-free hardware-virtualized *delivery* of virtual interrupts (Section 5.1), the *sending* of them is not yet virtualized, making this a major pain point for virtualizing popular applications like Memcached. Besides Memcached, our RocksDB and LevelDB benchmarks are of a similar nature (multi-threaded in-memory databases) and exhibit the same sensitivity.

To confirm our hypothesis that Ombro’s overheads come almost exclusively from VM entry/exit overheads (Section 10.2) and that this is responsible for our three macrobenchmark outliers, we conducted an informal experiment where we modified vanilla Xen to add artificial (busy-wait) overhead after each VM exit. With this modification, vanilla Xen’s macrobenchmark results closely matched those of Ombro (the same benchmarks showed similar slowdowns).

We conclude, therefore, that Ombro’s overheads on these outliers are not of great concern, as the IPI virtualization problem is shared with vanilla Xen (and x86 virtualization in general). We explore this further in Appendix A.1, where we compare vanilla Xen with a bare-metal (unvirtualized) system; we observe that baseline Xen’s overheads on Memcached, RocksDB, and LevelDB (and others) far outweigh the additional impact of Ombro on Xen. We note also that Intel has announced plans to introduce hardware-accelerated *IPI virtualization* in future processors [12] to address this issue; we expect this will eliminate Ombro’s overheads on these workloads as it will eliminate the underlying VM exits.

As Appendix A.2 details, none of Ombro’s effective overhead is attributable to its compile-time security instrumentation (split stack, CFI, and SFI). This bodes well for the prospects of using the SVA approach and infrastructure to implement stronger security hardening, such as full memory safety [20, 43, 57], on hypervisors, as hypervisor execution evidently does not represent a sufficient fraction of system run-time to make instrumentation on it costly in absolute terms.

11 Related Work

Compiler-based virtual machines decouple the instruction set used to express computation from the instruction set implemented by the hardware. Ombro builds directly upon prior work with the Low Level Virtual Architecture [5, 17] and Secure Virtual Architecture [16], described in Section 2.

Previous work has enforced CFI and/or return address integrity on systems code, but none have enforced return address integrity for hypervisors. KCoFI [13] and KCFI [26] enforce control flow integrity on OS kernel code but do not provide return address integrity. Silhouette [58] provides CFI and a

protected shadow stack for application code running in privileged mode on embedded systems and inspired our attack surface analysis (Section 9). Kage [24] uses compiler-based techniques similar to Ombro’s to provide CFI and a protected shadow stack for an embedded real-time OS while splitting the kernel into trusted and untrusted layers. IskiOS [28] provides a protected shadow stack for the Linux kernel by utilizing Intel’s Memory Protection Keys feature [30] but does not enforce it on hypervisor code. Ombro could incorporate IskiOS’s technique in lieu of SVA’s MPX-based SFI enforcement option if Intel deprecates MPX as planned [31].

HyperSafe [51] enforces control flow integrity on hypervisor code and controls how the hypervisor configures the MMU to prevent attackers from corrupting page tables, hypervisor code pages, or CFI labels. Unlike Ombro, it does not provide return address integrity and is therefore susceptible to advanced ROP attacks [7, 8, 18, 27]. The HyperSafe authors designed and evaluated a shadow stack variant in conjunction with their CFI scheme but found it to have high overhead (over 300%) due to its reliance on the x86 platform’s WP bit as an isolation mechanism [51]. In contrast, Ombro’s use of SFI for kernel-mode isolation provides efficient control stack protection. HyperSafe also does not appear to constrain VMX features (e.g., by protecting the VMCS) whereas Ombro does.

Whole-VM trusted execution environments (TEEs) that protect guest VMs from compromised hypervisors, such as CloudVisor [41, 56], H-SVM [32, 33], and HyperCoffer [55], are an orthogonal approach to hardening the hypervisor itself against attack. Ombro’s SVA-based VISC approach could readily lend itself to implementation of a whole-VM TEE using SFI and vISA restrictions in lieu of the hardware-based isolation mechanisms used by prior work, similar to Virtual Ghost’s [14] and Apparition’s [22] approach for OS kernels. In such a system, a whole-VM TEE could be *combined* with Ombro-style hypervisor hardening for additional defense-in-depth with (we believe) minimal cumulative overhead.

12 Future Work and Conclusions

Several interesting directions for future work exist. We can explore additional security policies that SVA-based systems could enforce on hypervisor code, such as code pointer integrity [38] and memory safety [20, 43, 57]. We can also explore replacing hardware-enforced security enforcement and isolation with compiler instrumentation techniques. For example, we could explore whether compiler-based enforcement could allow us to create hypervisors with isolated components as previous work [53] does using hardware isolation features.

In conclusion, we have expanded the SVA vISA and ported the Xen hypervisor to it, have used the vISA to implement the first protected shadow (split) stack for a hypervisor, and have demonstrated its efficiency on real-world benchmarks.

We thank the anonymous reviewers and shepherd for their helpful feedback. This work was supported by NSF grant CNS-1629770 and ONR award N00014-17-1-2996.

References

- [1] The Rust Programming Language. <https://www.rust-lang.org> [Online; accessed 2022-06-08].
- [2] The LLVM Compiler Infrastructure Project. [Online; accessed 2022-01-12].
- [3] Xen Project. <https://xenproject.org> [Online; accessed 2021-08-07].
- [4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information Systems Security*, 13:4:1–4:40, November 2009.
- [5] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVA: A Low-Level Virtual Instruction Set Architecture. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-36, pages 205–216, San Diego, CA, 2003. IEEE Computer Society.
- [6] Daniel P. Bovet and Marco Cesati. *Understanding the LINUX Kernel*. O'Reilly, Sebastopol, CA, 2nd edition, 2002.
- [7] Nicholas Carlini and David Wagner. ROP Is Still Dangerous: Breaking Modern Defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, August 2014. USENIX Association.
- [8] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (SEC)*, pages 161–176, Washington, D.C., 2015.
- [9] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium (SEC)*, pages 12–12, Baltimore, MD, 2005.
- [10] Clang Documentation. SafeStack. <https://clang.llvm.org/docs/SafeStack.html> [Online; accessed 18-June-2021].
- [11] Intel Corporation. Intel Xeon Silver 4208 Processor. <https://ark.intel.com/content/www/us/en/ark/products/193390/intel-xeon-silver-4208-processor-11m-cache-2-10-ghz.html>. [Online; accessed 2021-08-11].
- [12] Intel Corporation. Intel Architecture Instruction Set Features and Future Extensions Programming Reference. [https://software.intel.com/content/www/us/en/develop/download/intel-architecture-](https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html)
[instruction-set-extensions-programming-reference.html](https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html), May 2021. [Downloaded 2022-01-12].
- [13] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pages 292–307, San Jose, CA, May 2014.
- [14] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'14, pages 81–96, 2014.
- [15] John Criswell, Nicolas Geoffray, and Vikram Adve. Memory Safety for Low-Level Software/Hardware Interactions. In *Proceedings of the 18th USENIX Security Symposium*, Security'09, pages 83–100, 2009.
- [16] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP'07, pages 351–366, Stevenson, WA, 2007. ACM.
- [17] John Criswell, Brent Monroe, and Vikram Adve. A Virtual Instruction Set Interface for Operating System Kernels. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 26–33, Boston, MA, USA, June 2006.
- [18] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, August 2014. USENIX Association.
- [19] Hoss Firooznia (Universitato de Roĉestro Esperantistoj). About Esperanto. <https://esperanto.lodestone.org/esperanto/en> [Online; accessed 2022-06-08].
- [20] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006.
- [21] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. Spectres, Virtual Ghosts, and

- Hardware Support. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP'18, pages 5:1–5:9, Los Angeles, CA, 2018. ACM.
- [22] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. Shielding Software from Privileged Side-Channel Attacks. In *Proceedings of the 27th USENIX Security Symposium*, Security'18, pages 1441–1458, 2018.
- [23] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. pages 164–177, Bolton Landing, NY, USA, October 2003.
- [24] Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J Walls, and John Criswell. Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage. In *Proceedings of the 31st USENIX Security Symposium*, Security '22. USENIX Association, 2022.
- [25] Isaac Evans, Fan Long, Ulzii Bayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiropoulos-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 901–913, Denver, CO, 2015. ACM.
- [26] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 179–194, Saarbrücken, Germany, March 2016.
- [27] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pages 575–589, San Jose, CA, May 2014.
- [28] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. Fast Intra-Kernel Isolation and Security with IskiOS. In *Proceedings of the Twenty Fourth International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '21, 2021.
- [29] Hong Hu, Shweta Shinde, Sendroui Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.
- [30] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. May 2018. 325462-067US.
- [31] Intel Corp. Introduction to Intel® Memory Protection Extensions, July 2013. <https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-memory-protection-extensions.html> [Online; accessed 2020-11-10].
- [32] S. Jin, J. Ahn, J. Seol, S. Cha, J. Huh, and S. Maeng. H-SVM: Hardware-Assisted Secure Virtual Machines under a Vulnerable Hypervisor. *IEEE Transactions on Computers*, 64(10):2833–2846, Oct 2015.
- [33] Seongwook Jin, Jeongseob Ahn, Sanghoon Cha, and Jaehyuk Huh. Architectural Support for Secure Virtualization under a Vulnerable Hypervisor. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 272–283. IEEE, 2011.
- [34] Ethan Johnson, Komail Dharsee, and John Criswell. Secure Guest Virtual Machine Support in Apparition. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, pages 17–30, New York, NY, USA, 2019. ACM.
- [35] Mateusz Jurczyk and Sergei Glazunov. Google Project Zero: In-the-Wild Series: Windows Exploits. <https://googleprojectzero.blogspot.com/2021/01/in-wild-series-windows-exploits.html>, 2021. [Online; accessed 2021-11-02].
- [36] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: The Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, Ottawa, Ontario, Canada, Jun 2007.
- [37] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, SP'19, San Francisco, CA, 2019. IEEE.
- [38] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, Berkeley, CA, USA, 2014. USENIX Association.
- [39] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO'04, pages 75–86, Palo Alto, CA, 2004. IEEE Computer Society.

- [40] Phoronix Media. Phoronix Test Suite. <https://www.phoronix-test-suite.com>. [Online; accessed 2019-03-11].
- [41] Zeyu Mi, Dingji Li, Haibo Chen, Binyu Zang, and Haibing Guan. (Mostly) Exitless VM Protection from Untrusted Hypervisor through Disaggregated Nested Virtualization. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1695–1712. USENIX Association, August 2020.
- [42] Microsoft. Introduction to Hyper-V on Windows 10, 2018. <https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/> [Online; accessed 2021-08-07].
- [43] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09*, pages 245–258, New York, NY, USA, 2009. ACM.
- [44] OpenBenchmarking. Common Kernel Benchmarks. <https://openbenchmarking.org/suite/pts/kernel>. [Online; accessed 2021-08-11].
- [45] OpenBenchmarking. Phoronix Test Suite Documentation. <https://github.com/phoronix-test-suite/phoronix-test-suite/blob/master/documentation/phoronix-test-suite.md>. [Online; accessed 2022-01-12].
- [46] Oracle Corporation. VirtualBox. <https://www.virtualbox.org> [Online; accessed 2022-06-08].
- [47] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information Systems Security (TISSEC)*, 15(1):2:1–2:34, March 2012.
- [48] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Conference on Security, SEC'11*, pages 25–25, Berkeley, CA, USA, 2011. USENIX Association.
- [49] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the Expressiveness of Return-into-libc Attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, pages 121–141, Menlo Park, CA, 2011.
- [50] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles, SOSP'93*, pages 203–216, Asheville, NC, 1993. ACM.
- [51] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*, pages 380–395, May 2010.
- [52] David A. Wheeler. SLOCCount. <http://www.dwheeler.com/sloccount/> [Online; accessed 2022-06-08].
- [53] Dan Williams, Yaohui Hu, Umesh Deshpande, Piush K. Sinha, Nilton Bila, Kartik Gopalan, and Hani Jamjoom. Enabling Efficient Hypervisor-as-a-Service Clouds with Ephemeral Virtualization. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2016*, New York, NY, USA, 2016. ACM.
- [54] Xen Project. Understanding the Virtualization Spectrum, 2014. https://wiki.xenproject.org/wiki/Understanding_the_Virtualization_Spectrum [Online; accessed 2021-08-05].
- [55] Yubin Xia, Yutao Liu, and Haibo Chen. Architecture Support for Guest-Transparent VM Protection from Untrusted Hypervisor and Physical Attacks. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 246–257. IEEE, 2013.
- [56] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11*, pages 203–216, New York, NY, USA, 2011. ACM.
- [57] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19*, pages 631–644, 2019.
- [58] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J. Walls. Silhouette: Efficient Protected Shadow Stacks for Embedded Systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1219–1236. USENIX Association, August 2020.

- [59] Philipp Zieris and Julian Horsch. A Leak-Resilient Dual Stack Scheme for Backward-Edge Control-Flow Integrity. In *13th ACM Asia Conf. on Computer & Communications Security (ASIACCS)*, Incheon, Republic of Korea, June 2018.

A Additional Benchmarks

In this appendix, we provide results and discussion of additional benchmarks that we conducted in order to shed light on the main results presented in Section 10. We did not include these results in the main body of the paper because they either served to demonstrate existing issues known from prior work that we do not claim as novel results, or presented no statistically significant contrast and thus could be summarized adequately in the main text without detailed results.

A.1 Unmodified Xen vs. Unvirtualized

In Section 10.1, we observed that a few benchmarks from our subset of the Phoronix suite, particularly Memcached, RocksDB, and LevelDB, exhibited non-negligible overheads (unlike most of our macrobenchmarks) under Ombro as compared to baseline (“vanilla”) Xen. Through further experimentation (Section 10.2) and reference to existing literature in the field [41], we were able to conclude (Section 10.3) that the primary cause of these particular benchmarks’ poor behavior was likely due to the fact that they frequently send inter-processor interrupts (IPIs) to communicate between threads running on different virtual CPUs (vCPUs), which on current Intel processors requires taking a VM exit [12, 30].

As VM exits are expensive (slow) operations for hypervisors to handle due to the substantial amount of processor state that must be saved and loaded during a world switch, hypervisor and hardware design generally tries to make them infrequent as a share of total execution time. Workloads that do not make that possible (such as these problematic benchmarks) can therefore expect to incur substantial overheads simply from being virtualized in the first place.

To illustrate this issue, we re-ran our macrobenchmark suite (Section 10.1) on our test machine in a “bare-metal” (i.e. unvirtualized) configuration and compared the results to that of vanilla Xen, as shown in Table 4. For consistency with our main results, vanilla Xen is retained as the baseline, with the bare-metal results compared to it, yielding negative “overheads”, i.e. speedups, for bare metal. As can be seen, the benchmarks that showed non-negligible overheads under Ombro in Section 10.1 all run substantially faster on bare metal than under vanilla Xen, the difference between the two being far greater in magnitude than the difference between Ombro and vanilla Xen. In fact, most of the benchmarks, including the ones to which Ombro added no or negligible overhead over vanilla Xen, show significant gaps between unvirtualized and virtualized execution.

Our methodology for the bare-metal benchmark runs was to boot a copy of the same Linux installation used for the *domU* (unprivileged guest domain) in Xen-based benchmark runs directly from the system bootloader instead of through Xen. As it is the same system except for the addition of a few driver packages needed to support running on physical hardware, this minimizes differences between the configurations,

but the comparison is nonetheless imperfect: the bare-metal runs had access to the system’s full 32 GB of RAM, whereas the Xen-based runs were limited to 24 GB in the *domU*, as we needed to leave some of the system’s memory for Xen and the *dom0*. This, as well as effects such as the difference between virtualized and unvirtualized disk I/O, could potentially amplify the difference measured between vanilla Xen and bare metal. We therefore do not attempt to draw strong quantitative conclusions from the detailed results of this comparison and suffice to note qualitatively that the outliers in our Ombro benchmarks are clearly especially difficult cases for vanilla Xen as well. This is consistent with the conclusions of prior work [41] and the fact that Intel is planning to introduce *IPI virtualization* to future processors so that workloads such as these no longer need to incur frequent VM exits [12].

A.2 Ombro without Instrumentation

As part of our performance evaluation (Section 10), we wished to measure whether Ombro’s control flow integrity (CFI), software fault isolation (SFI), and split stack transformations had measurable impacts on performance. This would allow us to separate overheads due to instrumentation from those incurred simply by porting Xen to the SVA virtual instruction set (vISA).

To this end, we set a flag in the compiler that instructed it to not add CFI and SFI checks to the generated code or to perform the split stack transformation when building Xen in the Ombro configuration. This results in a build of Ombro which does not have functional hardening protections beyond vanilla Xen but still uses SVA intrinsics rather than native assembly to perform low-level operations. Thus, it measures any overheads from extra data copying or code indirection entailed by routing through the virtual instructions as well as the overheads of any runtime security checks performed by the intrinsics themselves.

We ran our macrobenchmark suite on this “no-instrumentation” build of Ombro/Xen and compared it with vanilla Xen, as summarized in Table 5. As in Appendix A.1, we use vanilla Xen as our baseline for consistency with our main results in Section 10.1. The overheads listed for “Ombro without instrumentation” can therefore be compared head-to-head with the “Ombro” numbers in Section 10.1.

As can be seen, the results for Ombro without instrumentation do not exhibit a clear contrast from the Ombro results in Section 10.1 that rises above the noise floor. (In fact, the geometric mean for Ombro without instrumentation shows *higher* overhead than ordinary Ombro, which can be clearly attributed to experimental noise given the high standard deviations on the benchmarks that turned out the most favorably for ordinary Ombro, particularly the PostgreSQL benchmarks.) We therefore conclude that the CFI and SFI instrumentation and the split stack transformation add no measurable overhead to the “core” vISA port. This makes sense in light of our conclusion from Sections 10.2 and 10.3 that Ombro’s overheads

are driven primarily by an increase in VM entry/exit latency, not by overheads on Xen’s own execution (e.g. scheduling and VM-exit handling such as hardware emulation).

The observation that CFI, SFI, and split-stack enforcement on Xen do not measurably impact overall performance indicates that Ombro’s increased VM entry/exit latency is coming from the extra data copying and operations performed by SVA’s implementation of VM entry/exit, rather than from Xen itself being slowed down by the compile-time security transformations. As our Ombro benchmarks were conducted with SVA’s standard bitmasking SFI implementation selected (Section 2.2) instead of the optional MPX-accelerated SFI implementation from Apparition [22] (which is in principle faster), this leads to a secondary conclusion that optimizing SVA’s CFI and SFI instrumentation is neither necessary nor worthwhile for Ombro, even though it has been for past SVA-based systems.

B Implementation and Porting Experience

Section 8 describes our prototype implementation of Ombro (i.e. the SVA compiler and runtime library plus our port of the Xen hypervisor to the SVA vISA) as used in our performance evaluation (Section 10). However, as the construction of this prototype represents a great deal of the work involved in this project, this appendix discusses that experience further for the benefit of interested readers. We discuss practical observations from the experience of building the prototype (B.1) as well as how our observations of the prototype’s performance fed back into the design process (B.2).

B.1 Engineering Observations

The process of porting Xen to SVA took two programmers (a PhD student and a research programmer) roughly two years to complete. This includes the substantial infrastructural improvements to SVA described in Section 8; qualitatively, we improved SVA from a minimally functional research prototype (previous SVA systems could not even run large-scale applications like the Apache web server without crashing) to one that, while perhaps not “production-grade”, can confidently run a complete Xen system with multiple guests hosting a full spectrum of complex end-user applications. These include a fully working MATE desktop GUI under the Ubuntu-based Linux Mint and a Wayland-based window manager (Sway) under Arch Linux (we used Arch for our benchmarks but exercised both distributions heavily during development); web browsers like Firefox and Chrome; development tools like Clang and GCC; and a full Phoronix suite of kernel-intensive real-world macrobenchmarks (Section 10). Notably, we did not need to exclude *any* benchmarks for lack of compatibility with Ombro, although a few failed to compile on the unmodified baseline system, likely due to the system compiler being too new.

We expect that, particularly with these SVA infrastructure improvements in place, an experienced hypervisor developer

could repeat our port of Xen (or port another hypervisor) to SVA/Ombro in substantially less time than we took, as we spent a lot of those two years learning about hypervisor and VMX inner workings and debugging opaque low-level issues.

Once we completed the port of Xen to SVA and had a fully working system, it was relatively straightforward to implement return address protection via a compile-time split stack transformation (Section 7). The split-stack-related changes to the compiler, SVA runtime library, and Xen were relatively small (see LOC numbers in Section 8) and took only about a month to complete, during which we also began the performance evaluation and writing of the paper.

We believe this success highlights the power and flexibility of the SVA approach. SVA provides a well-defined interface for low-level software/hardware interactions along with a robust toolkit of primitives useful for enforcing confidentiality and integrity policies, such as kernel-mode memory protection and mediation of hardware data structures (e.g. page tables and VMCSes). This foundation gives security researchers a proven framework within which they can experiment with novel security policies and enforcement mechanisms, allowing them to focus on the security and performance tradeoffs of their contributions rather than reinventing and reimplementing solutions to the myriad known pitfalls of securing kernel-mode code.

B.2 Performance-Driven Design Changes

As our performance analyses (Section 10 and Appendix A) show, the entirety of Ombro’s statistically significant overhead (only seen on certain IPI-heavy benchmarks) comes from the basic port of Xen to the SVA vISA, not from its compile-time security instrumentation on Xen (CFI, SFI, and split stack). In earlier versions of our vISA design, its overheads were substantially larger, and appeared on more benchmarks, than in the final version presented in this paper. These overheads yielded insights that drove several design changes to the vISA which dramatically improved performance to bring Ombro more in line with a non-SVA baseline.

Active vs. Loaded VMCSes in the vISA. The first such change, detailed in Section 5.3, corrected a limitation of the Shade [34] version of the vISA that was not previously evident because Shade’s hypervisor support was not sufficiently sophisticated to support a real-world hypervisor or evaluate performance at any level higher than microbenchmarks that exercised the functionality of individual VMX instructions. Shade interpreted Intel’s concept of there only being one “active VMCS” at a time on a processor [30] as meaning that a previous VMCS had to be explicitly unloaded (using the VMCLEAR instruction) before a different one could be loaded (via VMPTRLD). While porting Xen, however, it quickly became clear that Xen expects to be able to maintain multiple *loaded* VMCSes even as it switches between different *active* ones in vCPU context switches—i.e. to perform subsequent

VMPTLDRs on multiple VMCSes within a working set without VMCLEARING any of them until the respective vCPU is ready to be torn down or migrated to a different physical CPU.

To see how much impact Shade’s more restrictive interface would have on real-world performance, we modified Xen to explicitly flush the outgoing vCPU’s VMCS on every context switch and informally measured the impact on a *domU* guest running a context-switching stress test program we had created for debugging. If the (single-threaded) guest was the only significant load on the machine, no slowdown was evident compared to vanilla Xen—unsurprising, since few context switches were occurring. However, when we forced context switches by running a CPU-intensive application on all cores in the *dom0*, we found that the *domU*’s performance dropped precipitously, by over 4x.

Clearly, the hardware is able to take significant advantage of Xen’s default behavior by retaining multiple VMCSes in on-chip cache across vCPU context switches. Ombro therefore (Section 5.3) improves the vISA to support calling the `loadvm` intrinsic on a new VMCS without having to first call `unloadvm` on a then-current one. While this (slightly) complicates the vISA’s conceptual model of VMCS behavior compared to Shade, the performance improvements are well worth it, demonstrating that this facet of the native ISA is indeed essential to preserve at the vISA level.

Tying Guest State to SVA Thread Switches vs. VM Entry/Exit. The sole remaining source of vISA overhead that appeared in our benchmarks (Section 10 and Appendix A) is attributable to overhead on VM entry and exit. While entry/exit overheads should ideally not be performance-critical due to hardware-accelerated virtualization that makes VM exits rare, real systems fall short of this ideal. As our performance analysis in Section 10.3 explains, this is particularly true for IPI-heavy workloads that incur frequent VM exits. It is therefore desirable to minimize the SVA vISA’s impact on entry/exit latency as much as possible.

As Section 5.2 discusses, Shade’s version of the `runvm` intrinsic [34] was designed to present a conceptually clean abstraction wherein no guest state is ever active on the processor when running in host mode (outside of the `runvm` intrinsic itself) or vice versa. This necessitated that `runvm`’s implementation context-switch *all* system state elements, on every entry and exit, that could be modified by a guest and which could affect host software’s view of system state.

While porting Xen, we realized that this design decision was overly prescriptive on hypervisor and host-OS design and negatively impacted performance, since it forced heavy-weight state components such as the FPU and MSRs to be switched on every VM entry/exit. In practice, the hypervisor/kernel is not expecting SVA to completely hide the guest’s existence from it; rather, it will save and restore its own state on higher-level context switches (between guest vCPUs or host userspace threads) to accommodate the guest’s occupation of the processor. This allows the hypervisor/kernel to

minimize unnecessary copying by refraining from disturbing major state components like the FPU except when it decides to schedule a different vCPU or thread to run; it also allows it to implement lazy FPU saving [6] if desired.

Since its original versions [16, 17], SVA has provided vISA primitives to assist OS kernels in safely transitioning between their own execution state and that of userspace threads as they handle interrupts, traps, and system calls and make context switches. This, we realized, is exactly the model used by real-world hypervisors for making context switches between vCPUs. It was therefore natural to eliminate Shade’s excessive orthogonality between userspace-thread and guest-vCPU context switches in favor of unifying them under SVA’s existing *thread* abstraction [13, 14]. As described in Section 5.2, guest vCPU state is now stored in the same fields within SVA’s thread context structures as used for userspace threads, the only difference being that the hypervisor/OS kernel chooses to enter that context via a call to the `runvm` intrinsic (VM entry to VMX non-root mode) rather than via SVA’s `iret` function (return to host Ring 3 from interrupt/trap/syscall handler).

Besides streamlining the SVA vISA, this conceptual change improved Ombro’s VM entry/exit overhead from over 200% to just 95% (Section 10.2) and macrobenchmark overhead on the worst IPI-heavy outlier from 60% to 21.81% (Section 10.1). Based on further informal experimentation, we believe reducing the overhead further may be possible, but the current implementation has reached a point of diminishing returns, making it more profitable to focus on eliminating the root cause of excessive VM exits in the outlier benchmarks, e.g. through Intel’s upcoming hardware IPI virtualization support [12] as discussed in Section 10.3 and Appendix A.1.

C Background on the Name *Ombro*

For readers who are curious what the name *Ombro* signifies, it is a word meaning “shade” or “shadow” in the constructed international auxiliary language Esperanto [19]—the only artificial language that has successfully become a “living” human language. This follows a loose tradition within our research group of naming systems after a general theme of “ghosts” or “shadows”, which originated with systems building on the Virtual Ghost [14] project. These initially included Apparition [22] and Shade [34] (direct descendants of Virtual Ghost) and expanded to include some non-SVA-based shadow stack projects such as Silhouette [58], IskiOS [28], and Kage [24] (the latter two translating “shadow” in Greek and Japanese respectively). *Ombro* was doubly appropriate for the work presented in this paper as it can be interpreted either as a translation of “Shade” (the project we directly extend) or as referring to the “shadow stacks” we provide for Xen to protect return addresses.

The Esperanto origin of *Ombro* is also apropos to the nature of a virtual instruction set computing (VISC) system like SVA, as Esperanto is an academically-constructed yet practically-purposed human language designed to streamline

second language learning by minimizing irregularities and exceptions—much as the SVA virtual instruction set architecture (vISA) does in relation to native hardware ISAs to make analysis and protection of low-level system software easier. Like SVA’s role in mediating the interface between hardware ISAs and low-level software, Esperanto is not meant to *replace* native languages but to supplement them in situations where they struggle to fulfill their communication goals.

The *Ombro* name was selected by the lead author (Ethan Johnson) who studied and learned Esperanto as a hobby during the development of this work. He has attained intermediate proficiency in the language and welcomes correspondence either in English *ăŭ en Esperanto*.

Table 4: Unmodified Xen 4.12.0 vs. No Hypervisor (arrows indicate whether higher or lower is better)

Benchmark	Units	Vanilla Xen	Std. Dev.	Bare Metal	Std. Dev.	% Ovhd.
RocksDB	ops/s ↑	34557.667	0.1%	852098.333	1.1%	-2365.73%
Memcached	ops/s ↑	46960.733	0.2%	84270.000	0.6%	-79.45%
PostgreSQL	TPS ↑	4266.496	26.7%	7625.012	7.8%	-78.72%
Hackbench	s ↓	140.056	0.8%	37.645	4.2%	-73.12%
LevelDB	μs/op ↓	428.367	0.1%	168.999	0.2%	-60.55%
IPC (TCP Socket)	messages/s ↑	473697.400	13.3%	739706.667	2.2%	-56.16%
PostgreSQL	ms (avg. lat.) ↓	61.644	21.0%	32.989	8.5%	-46.48%
Apache	req/s	212698.913	0.4%	257396.147	0.3%	-21.01%
OSBench (Create Files)	μs/event ↓	24.270	0.3%	19.570	0.1%	-19.37%
PostMark	TPS ↑	4335.000	1.0%	5102.000	0.0%	-17.69%
pmbench	μs ↓	0.113	4.5%	0.094	1.0%	-16.54%
MBW	MiB/s ↑	6694.581	0.3%	7793.890	0.8%	-16.42%
ctx_clock	clocks ↓	6694.581	0.3%	7793.890	0.8%	-16.42%
t-test1	s ↓	23.587	0.9%	21.874	0.5%	-7.26%
iPerf (TCP)	Mbits/s ↑	28029.667	2.3%	29551.667	0.3%	-5.43%
SQLite	s ↓	83.164	0.8%	81.030	0.6%	-2.57%
BenchmarkMutex	ns ↓	39.967	0.9%	39.200	0.0%	-1.92%
StressNG (RdRand)	bogo ops/s ↑	250298.447	0.0%	251830.763	0.0%	-0.61%
OpenSSL (RSA 4096)	signs/s ↑	1516.867	0.3%	1499.200	0.6%	1.16%
Schbench	μs ↓	18762.667	0.7%	20819.200	3.1%	10.96%

Table 5: Unmodified Xen 4.12.0 vs. Ombro without CFI, SFI, or Split Stack Transformations (arrows indicate whether higher or lower is better)

Benchmark	Units	Vanilla Xen	Std. Dev.	Ombro without instrumentation	Std. Dev.	% Ovhd.
PostgreSQL	ms (avg. lat.) ↓	61.644	21.0%	59.805	19.9%	-2.98%
MBW	MiB/s ↑	6694.581	0.3%	6884.724	0.2%	-2.84%
pmbench	μs ↓	0.113	4.5%	0.110	3.8%	-2.60%
PostgreSQL	TPS ↑	4266.496	26.7%	4364.444	24.1%	-2.30%
OSBench (Create Files)	μs/event ↓	24.270	0.3%	24.207	0.1%	-0.26%
BenchmarkMutex	ns ↓	39.967	0.9%	39.933	1.0%	-0.08%
OpenSSL (RSA 4096)	signs/s ↑	1516.867	0.3%	1517.400	0.1%	-0.04%
SQLite	s ↓	83.164	0.8%	83.152	0.8%	-0.01%
ctx_clock	clocks ↓	240.000	0.0%	240.000	0.0%	0.00%
PostMark	TPS ↑	4335.000	1.0%	4335.000	1.0%	0.00%
StressNG (RdRand)	bogo ops/s ↑	250298.447	0.0%	250127.343	0.0%	0.07%
t-test1	s ↓	23.587	0.9%	23.642	1.1%	0.23%
Apache	req/s	212698.913	0.4%	208431.403	0.1%	2.01%
iPerf (TCP)	Mbits/s ↑	28029.667	2.3%	27447.667	2.4%	2.08%
Schbench	μs ↓	18762.667	0.7%	19934.857	9.8%	6.25%
Hackbench	s ↓	140.056	0.8%	149.015	0.5%	6.40%
IPC (TCP Socket)	messages/s ↑	473697.400	13.3%	430804.400	8.6%	9.05%
LevelDB	μs/op ↓	428.367	0.1%	494.449	0.1%	15.43%
RocksDB	ops/s ↑	34557.667	0.1%	28721.333	0.4%	16.89%
Memcached	ops/s ↑	46960.733	0.2%	35734.733	1.5%	23.91%
Geometric Mean						3.32%



HyperEnclave: An Open and Cross-platform Trusted Execution Environment

Yuekai Jia¹, Shuang Liu², Wenhao Wang^{3,4(✉)}, Yu Chen¹, Zhengde Zhai²,
Shoumeng Yan², and Zhengyu He²

¹Tsinghua University

²Ant Group

³SKLOIS, Institute of Information Engineering, CAS

⁴School of Cyber Security, University of Chinese Academy of Sciences

Abstract

A number of trusted execution environments (TEEs) have been proposed by both academia and industry. However, most of them require specific hardware or firmware changes and are bound to specific hardware vendors (such as Intel, AMD, ARM, and IBM). In this paper, we propose HyperEnclave, an open and cross-platform process-based TEE that relies on the widely-available virtualization extension to create the isolated execution environment. In particular, HyperEnclave is designed to support the *flexible enclave operation modes* to fulfill the security and performance demands under various enclave workloads. We provide the enclave SDK to run existing SGX programs on HyperEnclave with little or no source code changes. We have implemented HyperEnclave on commodity AMD servers and deployed the system in a world-leading FinTech company to support real-world privacy-preserving computations. The evaluation on both micro-benchmarks and application benchmarks shows the design of HyperEnclave introduces only a small overhead.

1 Introduction

In recent years, trusted execution environments (TEEs) are emerging as a new form of computing paradigm, known as confidential computing, due to the high demand for privacy-preserving data processing technologies that can handle massive data samples. TEEs provide hardware-enforced memory partitions where sensitive data can be securely processed. Existing TEE designs support different levels of TEE abstractions, such as process-based (Intel's Software Guard eXtensions (SGX) [55]), VM-based (AMD SEV [45]), separate worlds (ARM TrustZone [16]), and hybrid (Keystone [49]). Currently, the most prominent example of TEEs is Intel SGX, which is widely available in commercial off-the-shelf (COTS) desktop and server processors.

Motivations. Most of today's TEE technologies are close-sourced and require specific hardware or firmware changes

that are difficult to audit, slow to evolve, and thus are inferior to cryptographic alternatives (such as homomorphic encryption), which are based upon public algorithms and widely available hardware. Moreover, most existing TEE designs restrict the enclaves (i.e., the protected TEE regions) to run only in fixed mode.¹ It is difficult to support the performance and security requirements of various types of applications that need to be protected by TEEs. For example, Intel SGX enclaves run in the user mode and cannot access privileged resources (such as the file system, the IDT, and page tables) and process privileged events (interrupt and exceptions). As a result, running I/O-intensive and memory-demanding tasks leads to significant performance degradation.

To fill the gap, in this paper we propose the design of HyperEnclave to support *confidential cloud computing* that can run securely on both legacy servers readily available in the cloud, and on the rising ARM (or RISC-V in the future) servers, without requiring specific hardware features. For this purpose, our design provides a process-based TEE abstraction using the widely available virtualization extension (for isolation) and TPM (for root of trust and randomness etc.). To better fulfill the needs for specific enclave workloads, HyperEnclave supports the *flexible enclave operation modes*, i.e., the enclaves can run at different privilege levels and can have access to certain privileged resources (see Sec. 4 for more details).

Design details. In our design, the system runs in three modes. A trusted software layer, called RustMonitor (security monitor written in Rust), runs in the *monitor mode*, which is mapped to the VMX root mode. RustMonitor is responsible for enforcing the isolation and is part of the trusted computing base (TCB). The untrusted OS (referred to as the *primary OS*) provides an execution environment for the untrusted part of applications; the untrusted OS and application parts run in the *normal mode*, which is mapped to the VMX non-root mode. The trusted part of application (i.e., *enclave*) runs in the *secure mode*, which can be mapped *flexibly* to ring-3 or ring-0 of the VMX non-root mode, or ring-3 of the VMX root mode.

¹An exception is CURE [20], which however requires hardware changes to the CPU core and the system bus to support the flexible enclaves (Sec. 9).

*Corresponding author: Wenhao Wang (wangwenhao@ie.ac.cn).

Memory isolation is enforced with hardware-based memory protection of the memory-management unit (MMU). As we observe that existing process-based TEEs (e.g., Inktag [38] and Intel SGX [55]) are vulnerable to page-table-based attacks [74], our memory isolation scheme chooses to manage the enclave’s page table and page fault events entirely by the trusted code, removing the involvement of the primary OS. The design also prevents certain types of enclave malware attacks (Sec. 3.2).

To minimize the attack surface, we adopt an approach called *measured late launch*: the primary OS kernel is first booted; then a chunk of special kernel code, implemented as a kernel module in the primary OS, runs to initiate RustMonitor in the most privileged level (i.e., the monitor mode) and demotes the primary OS to the normal mode. All booted components during the booting process are measured and extended to the TPM Platform Configuration Registers (PCRs). Since the TPM attestation guarantees that PCRs cannot be rolled back, the design ensures that RustMonitor is securely launched; otherwise, a violation of the TPM *quote* would be detected during remote attestation.

We have implemented HyperEnclave on commodity AMD servers. In total RustMonitor consists of about 7,500 lines of Rust code. The APIs of our enclave SDK are compatible with the official SGX SDK. As a result, code written for SGX could be easily ported to run on HyperEnclave by recompiling the code with little (or no) source code changes. We have ported a number of SGX applications, as well as the Rust SGX SDK [71] and the Occlum library OS [64] to HyperEnclave. The micro-benchmarks show that the overheads for ECALLs and OCALLs are < 9,700 and < 5,260 cycles respectively (14,432 and 12,432 cycles respectively on Intel SGX). The evaluation on a suite of real-world applications shows that the overhead is small (e.g., the overhead on SQLite is only 5%).

Contributions. In summary, the paper proposes the design of HyperEnclave, with the following contributions:

- An open² and cross-platform process-based TEE with minimum hardware requirements (virtualization extensions and TPM) that can run existing SGX programs with little or no source code changes, which enables the reuse of the rich toolchains and ecosystem for Intel SGX.
- Supporting the flexible enclave operation modes to fulfill the diverse security and performance requirements of enclave applications without hardware or firmware changes.
- A memory isolation scheme that the enclave’s page table and page fault are managed entirely by the trust code, which mitigates the page-table-based attacks and the enclave malware attacks.
- A measured late launch approach, combined with the TPM-based attestation to reduce the attack surface.
- An implementation on commodity servers (mostly) using the memory safe language Rust, and an evaluation on real

hardware and applications, demonstrating that the proposed design is practical and only has a small overhead.

2 Background

2.1 Trusted Execution Environment

A Trusted Execution Environment (TEE) is designed to ensure that sensitive data is stored, processed, and protected in an isolated and trusted environment. The isolated area could be a separate system apart from the normal operating system (such as the TrustZone [16] secure world), a part of a process address space (such as an Intel SGX [55] enclave), or a stand-alone VM (such as a virtual machine protected by AMD SEV [45] or Intel TDX [41]). To resist the privileged attacker, TEE needs to thwart not only the OS-level adversary but also the malicious party who has physical access to the platform. To this end, it offers hardware-enforced security features including isolated execution, integrity, and confidentiality protection of the enclave, along with the ability to authenticate the code running inside a trusted platform through remote attestation.

Isolation. At the core of a TEE is the memory isolation scheme, which guarantees that code, data, and the runtime state of the enclave cannot be accessed or tampered with by untrusted parties. For Intel SGX, the protected memory (i.e., the *enclave*) is mapped to a special physical memory area called Enclave Page Cache (EPC), which is encrypted and cannot be directly accessed by other software, firmware, BIOS, and direct memory access (DMA).

Attestation. The goal of remote attestation is to generate an attestation *quote*, which includes the measurement of the software state, signed with the attestation key embedded in the hardware. The remote user verifies the validity of the quote by checking the signature (which reflects the hardware identity) and the measurement (which proves the software state).

2.2 Trusted Platform Module

Trusted Platform Module (TPM) is both an industry-standard [36] and an ISO/IEC standard [4] for a secure cryptoprocessor. It is used by nearly all PC and server manufacturers. Firmware TPMs (fTPMs) are firmware-based (e.g. UEFI) TPM implementations. At the time of this writing, Intel, AMD, and Qualcomm all have implemented fTPMs.

TPM has a set of Platform Configuration Registers (PCRs), which can be used for the measurement of the booted code during the boot process. PCRs are reset to zero on system reboot or power on-off. During every boot process, the PCRs can only be extended with the new measurement (called PCR extend), and thus cannot be set to arbitrary values.

Every TPM ships with a unique asymmetric key, called the Endorsement Key (EK), embedded by the manufacturer as the root of trust. The TPM can generate a quote of the PCR

²The code will be available at <https://github.com/HyperEnclave>.

values, signed using the TPM Attestation Identity Keys (AIK), while the AIK is generated inside TPM and certified using EK. Any modifications of the booted code would be reflected in the quote. Upon receiving the quote, the remote party can validate the signing key comes from an authentic TPM and can be assured that the PCR digest report has not been altered.

2.3 Threat Model

Like the other TEE proposals [23, 49], we trust the underlying hardware, including the processor establishing the virtualization-based isolation, the System Management Mode (SMM) code, as well as the TPM. We assume that the Core Root of Trust for Measurement (CRTM) is trusted and immutable. HyperEnclave mitigates certain physical memory attacks, such as cold boot attacks and bus snooping attacks with the hardware support for memory encryption. We don't fully trust the operator and assume the attacker cannot mount physical attacks during the boot process, i.e., we assume that the system is initially benign (during system boot), and the early OS during the boot stage is part of the TCB. This can be achieved in two ways.

- *Firstly*, the power-on event can be secured with a hardware device, such as an HSM (i.e., hardware security module). The platform enters the boot process only with the engagement and supervision of a trusted party, who owns the HSM. After that, the operators for maintenance are not trusted.
- *Secondly*, the boot process can be enhanced to defend against adversaries with physical accesses. To prevent I/O attacks, we can harden the OS to remove unnecessary devices and disable the DMA capability of peripherals before IOMMU is enabled. We can enable memory encryption at an early stage (e.g., in the BIOS, before any off-chip memory is used) to prevent physical memory attacks.

However, after RustMonitor is launched, the primary OS is demoted to the normal mode, and can be under the control of the attacker, who may try to compromise the RustMonitor or enclaves, e.g., try to access the protected memory directly or through DMA. We consider the enclave code may be malicious or controlled by an attacker due to memory bugs. Our design needs to prevent a compromised enclave from contaminating the other enclaves or the RustMonitor. We also prevent the attacks against the primary OS or the application code, such as those presented in [63]. Similar to other TEEs, in this paper we do not focus on the prevention of denial of service (DoS) attacks or side channel attacks, such as cache timing and speculative execution attacks [48].

3 Design

HyperEnclave is designed to support confidential cloud computing without requiring specific hardware features. Therefore, HyperEnclave is built upon the widely available virtual-

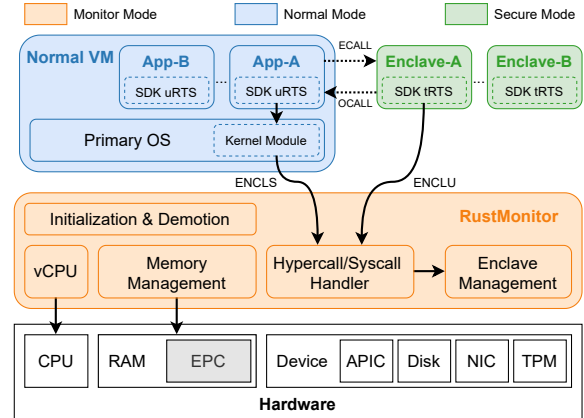


Figure 1: System Overview.

ization extension. In particular, HyperEnclave is designed to support the process-based TEE model (similar to Intel SGX) for the following reasons.

- *Minimized TCB.* To protect an application using the process-based TEE, the TCB includes only the protected code itself, while in the other forms of TEE, much more code must be included, such as the guest operating system for VM-based TEEs.
- *Established ecosystem.* Since Intel SGX is currently the most prevalent TEE supported in the cloud (major CSPs, including GCP, Azure, and Aliyun, provide SGX-based instances [9, 62]), a rich set of toolchains and applications have been developed. Supporting the SGX model reduces the porting effort and makes it easy to deploy confidential computing tasks in the cloud.
- *Cloud computing trends.* We have witnessed a clear trend towards running container-based serverless applications in the cloud. Protecting these applications against untrusted clouds using TEEs is important. Considering that such computing tasks are typically short-lived, and favor a short start-up time, maintaining a VM seems to be too heavy-weight.

In this section, we introduce HyperEnclave using x86 notations, as we prototyped HyperEnclave on AMD servers.

3.1 System Overview

HyperEnclave supports the following modes: the monitor mode, i.e., VMX root operation mode; the normal mode for the primary OS and untrusted part of applications, i.e., ring-0 and ring-3 of the VMX non-root operation mode respectively; and the secure mode for the enclave, which could be ring-3 and ring-0 of the VMX non-root operation mode, or ring-3 of the VMX root operation mode, depending on the *enclave operation mode*. We will introduce the flexible operation mode supported by HyperEnclave in Sec. 4. As illustrated in Figure 1, HyperEnclave consists of the following components:

- *RustMonitor* is a lightweight hypervisor running in the monitor mode that manages the enclave memory, enforces the memory isolation, and controls the enclave state transitions. It works as a resource monitor, while complicated tasks are offloaded to the primary OS.
- RustMonitor creates a *unique* guest VM (referred to as the *normal VM*) that runs the *primary OS* (such as Linux) and hosts the untrusted part of applications in the normal mode. The primary OS is still in charge of process scheduling and I/O devices management, but it is not trusted by the RustMonitor and enclaves.
- *Application* is the untrusted part of the application which runs in the primary OS.
- *The kernel module*. We provide a kernel module in the primary OS to load, measure, and launch RustMonitor, as well as to invoke the emulated privileged operations.
- To ease development, HyperEnclave provides an *enclave SDK* with APIs compatible with the official Intel SGX SDK [12], including both the untrusted runtime and trusted runtime (i.e., SDK uRTS and SDK tRTS). As such, most SGX programs can run on HyperEnclave with little or no source code changes.
- *Enclave* is the trusted part of the application running in the secure mode.

3.2 Memory Management and Protection

Challenges. For process-based TEEs, the enclave runs in the user mode and is not able to manage its own page table. Existing designs (e.g., Intel SGX, TrustVisor [54]) allow the untrusted OS to manage the enclave’s page table. To prevent memory mapping attacks (i.e., attacks by manipulating the enclave’s address mappings, as shown in Figure 9, Appendix A.1), the design of SGX extends the Page Missing Handler (PMH) and introduces a new metadata called EPCM for additional security checks on TLB misses [32]. Without secure hardware support, a prevalent software solution [19, 54, 75] is to make the page tables write-protected by setting the page table entries (PTEs) for pages holding the page tables, i.e., any update to the page table traps to the hypervisor and then be verified. However, on x86 platforms the updates of access and dirty bits of the PTEs also trap into the hypervisor, leading to non-negligible overhead. Even-worse, since the enclave page fault is also processed by the OS, the above designs are still vulnerable to the page table-based attacks, such as the controlled-channel attacks [74].

The design becomes more challenging to support enclave dynamic memory management (i.e., EDMM on SGX2 platforms [34]), i.e., dynamically adding or removing enclave pages, or changing the enclave page attributes or types after the enclave is initialized. Without EDMM, all physical memory that the enclave might ever use must be committed before enclave initialization. Therefore, EDMM reduces en-

clave build time and enables new enclave features, such as on-demand stack and heap growth, and on-demand creation of code pages to support just-in-time (JIT) compilation. On SGX2 platforms, the enclaves need to send the EDMM request to the SGX driver through OCALLs, who then makes the requested changes. Since the driver is untrusted by the enclaves, the changes need to be explicitly checked and accepted by the enclaves to take effect, which involves heavy enclave mode switches.

HyperEnclave memory management. We observe that the above challenges are rooted in the fact that the enclave’s page table and page faults are both managed by the primary OS. In HyperEnclave, though the enclave is still part of the application’s address space, we create a separate page table for the enclave and let RustMonitor manage the enclave’s page table and page fault without the involvement of the primary OS³, while the page tables in the normal VM are still managed by the primary OS. However, the design faces new challenges: since the enclave can access the application’s entire address space, upon a change to the mapping of the page tables in the applications, e.g. due to page swapping, the updated mapping needs to be synchronized to the enclave’s page table managed by RustMonitor.

To eliminate the overhead for synchronization, we pre-allocate a *marshalling buffer* in the application’s address space, which is shared with the enclave. The mappings of the marshalling buffer are fixed during the entire enclave life cycle by pre-populating the physical memory and pinning it in the memory. All data exchanged between the enclave and the application must be passed through the marshalling buffer. The application’s memory mappings (except those for the marshalling buffer) are not needed by the enclave and are not included in the enclave’s page table. Such a design also mitigates the known enclave malware attacks [63], as the enclave cannot access the application’s address space but the marshalling buffer (Sec. 6 for more details). We remind the attacker may manipulate the marshalling buffer, however it does not cause additional security issues, since the buffer is untrusted by design where the developer is responsible to ensure that the data transmitted through the buffer is authentic and protected (same as the SGX model).

When the enclave accesses a virtual address that is not committed with a physical page (e.g., due to page swapping or EDMM), a page fault is raised and the enclave traps to RustMonitor. RustMonitor picks up a free page from the enclave memory pool, inserts a new mapping to the enclave’s page table, and resumes the enclave’s execution. When the enclave requests changing the page permissions, the enclave issues a hypercall to RustMonitor to update the permissions in the enclave’s page table and clear the corresponding TLB entries.⁴

HyperEnclave memory isolation. Figure 2 shows the mem-

³P-Enclave can manage its own guest page table (Sec. 4.3).

⁴P-Enclave can change the page permissions by itself (Sec. 4.3).

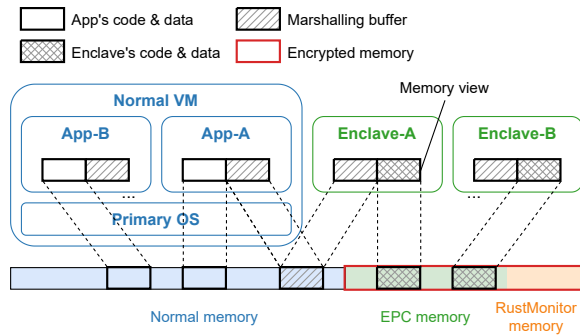


Figure 2: Memory isolation.

ory mappings of the applications within the normal VM and the enclaves. The application’s memory within the normal VM is managed with nested paging, while the enclave’s memory could be managed through nested paging or through normal 1-level address translation, determined by the corresponding operation mode (Sec. 4). As a result, HyperEnclave enforces the following security requirements.

- **R-1:** The primary OS and applications are not allowed to access the physical memory belonging to RustMonitor and the enclaves.
- **R-2:** The enclave is not allowed to access physical memory belonging to RustMonitor and other enclaves. It is designed to have access to only a specific memory region shared with the untrusted application for parameter passing (i.e., the marshalling buffer).
- **R-3:** DMA accesses from malicious peripherals to the physical memory belonging to RustMonitor and the enclaves are not allowed. In order to prevent such attacks, HyperEnclave restricts the physical memory used by the peripherals with the support of the Input-Output Memory Management Unit (IOMMU) in modern processors.

Memory encryption. To thwart physical memory attacks, such as cold boot and bus snooping attacks, HyperEnclave may leverage hardware memory encryption (such as AMD SME [44] and Intel MKTME [42]) to encrypt partial physical memory at the page granularity. If the platform does not support hardware memory encryption, HyperEnclave may consider to apply software approaches [76] to encrypt the isolated memory. This approach, however, may impose substantial overhead compared with hardware based solutions.

3.3 Trusted Boot, Attestation and Sealing

Measured Late Launch. The boot process of HyperEnclave is shown in Figure 3. On system boot, a static and immutable piece of code, known as the Core Root of Trust for Measurement (CRTM), executes first to bootstrap the process of building a measurement chain for subsequent firmware and software, including the BIOS, grub, the primary OS kernel, and initramfs. The measurements are stored to TPM PCRs

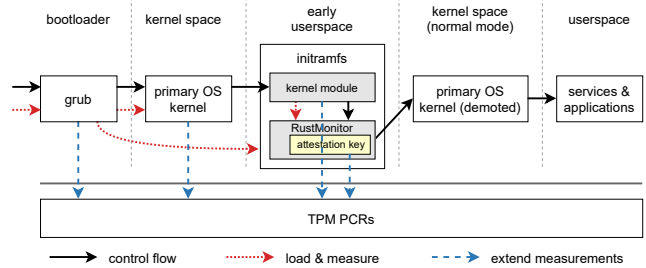


Figure 3: Measured Late Launch.

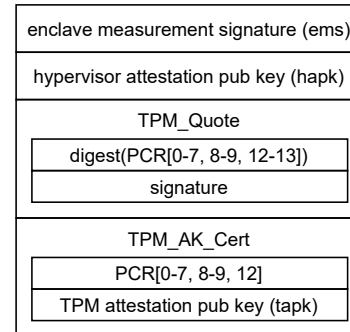


Figure 4: The HyperEnclave quote structure.

for each boot component, so that any modification will be reflected in the attestation quote.

To reduce the attack surface from the primary OS, we put the RustMonitor image into the initramfs. The kernel measures the RustMonitor image and extends the value to TPM PCRs, then it launches RustMonitor in *early userspace*, i.e., before any userspace program that relies on the disk file system starts to run. Along with the measured boot, it ensures that the software state when RustMonitor is loaded is trusted.

After RustMonitor is loaded, the execution continues at the pre-defined entry. RustMonitor sets up its own running context (such as the stack, page table, IDT, etc.) and prepares the virtual CPU (vCPU) configurations for each CPU. Then RustMonitor launches the normal VM and demotes the primary OS to the normal mode. Returning to the kernel module, the kernel continues to boot in the normal mode and is unaware of the existence of RustMonitor.

HyperEnclave applies the above approach (referred to as *measured late launch*) so that RustMonitor is loaded as a type-2 hypervisor (like KVM) while runs as a type-1 hypervisor (like Xen). In this way, RustMonitor does not need to trust the primary OS anymore after the primary OS is demoted to the normal mode.

Remote Attestation. With the measured late launch, all booted components are measured and extended to the TPM. After RustMonitor is booted, it needs to extend the trust to the enclaves. For this purpose, RustMonitor derives an attestation key pair which is used to sign the enclave measurement. Then RustMonitor extends the derived public key to the TPM

PCR, and the private key never leaves RustMonitor which is protected by memory isolation and encryption.

During enclave creation, all pages added to the enclave (including the corresponding page content, page type, and RWX permissions) are measured by RustMonitor to generate the enclave measurement. The (intermediate) measurement is stored in RustMonitor's memory, which is invisible to the enclaves and the primary OS.

Similar to TPM and Intel SGX, HyperEnclave adopts a SIGn-and-MAC (SIGMA) attestation protocol for the remote attestation flow. As shown in Figure 4, we denote the public key of RustMonitor's attestation key by the hypervisor attestation public key (`hapk`). The enclave measurement is signed using RustMonitor's attestation key to form the enclave measurement signature (`ems`). The TPM quote `TPM_Quote`, which is signed using the TPM attestation key, includes the PCRs for the measurement of all booted code, and the measurement of `hapk`. Upon receiving the attestation report, the remote user can verify the report by comparing the measurement of booted code (including the CRTM, BIOS, grub, kernel, initramfs, and hypervisor) and the enclave, as well as verifying the certificate chain for generating the signature.

Secret key generation. When RustMonitor is initialized for the first time, it generates a root key K_{root} from the random number generator (RNG) module of the TPM. K_{root} is stored outside the TPM using TPM's seal operation. During the booting process on system reset, RustMonitor decrypts K_{root} using TPM's unseal operation, which guarantees that K_{root} can only be unsealed with the exactly same TPM chip with matching PCR configurations. Furthermore, RustMonitor floods the PCRs with a constant before transferring control to the primary OS to prevent it from retrieving K_{root} . All other key materials, including the enclave's sealing key and report key are derived from both K_{root} and the enclave's measurement.

3.4 The Enclave SDK

Porting existing applications to the enclaves can be cumbersome since TEEs usually expose limited hardware and software interfaces and provide additional security services (e.g., attestation and sealing). For process-based TEEs, the applications need to be partitioned into the trusted and untrusted parts, and the interfaces need to be carefully designed to avoid various security pitfalls [27, 46, 69]. A lot of effort has been spent and many tools have been developed for Intel SGX, due to its dominant position in the market, including library OSes [64, 67], containers [18], automatic partition and protection tools [50, 68], WebAssembly Micro Runtime [57], and interface protection [65]. Consequently, Intel SGX has supported securely running applications written in C/C++, Rust, Java, Python, etc., without expensive code refactoring.

We provide the enclave SDK with APIs compatible with the official Intel SGX SDK to ease the development of applications on HyperEnclave. The enclave SDK is retrofitting the

official SGX SDK. By replacing the SGX user leaf functions (e.g., `EENTER`, `EEXIT`, and `ERESUME`) with hypercalls, SGX programs can run on HyperEnclave with little or no source code changes. Once the enclave executes these user leaf functions, it traps to RustMonitor and RustMonitor emulates the functionalities of the corresponding SGX instructions.

The enclave is compiled as a trusted library of the application, while the application itself runs in the primary OS. The enclave life cycle is managed through the emulation of a set of privileged SGX instructions (i.e., `ECREATE`, `EADD`, `EINIT`, etc.). To this end, the kernel module running in the primary OS provides similar functionalities by invoking RustMonitor through hypercalls, and exposes the functionalities to the applications by the `ioctl()` interfaces. By emulating the privileged SGX instructions, RustMonitor is responsible for the management of the enclave's life cycle (Sec. 4).

To be compatible with the official Intel SGX SDK, most data structures involved in HyperEnclave (such as the `SIGSTRUCT` structure, the `SECS` page, and the `TCS` page) are similar to that of SGX. With the HyperEnclave design, it is straightforward to support dynamic enclave management in an enclave, since the enclave memory and page fault are all managed by RustMonitor. Multi-threading within the enclave is supported by associating one `TCS` page for each enclave thread within the enclave. Exception handling within the enclave is supported by setting more than 1 `SSA` page for each `TCS`. The details are omitted due to space constraints and we refer the readers to the SGX manual [11] for more details.

4 Flexible Enclave Operation Mode

A wide range of existing applications can be offloaded to the TEEs, such as computing-intensive tasks (machine learning [60]), input and output (IO)-intensive tasks (such as the Apache and Nginx web server [18]), memory-intensive tasks (Redis and Memcached [18]), and tasks which favor in-enclave exception handling and privilege separation [21]. Most TEEs support running the enclaves only in fixed mode, Intel SGX (also TrustVisor [54] and Secage [51]) enclaves in particular, as part of the application address space, run in user mode. As a result, the user mode enclave is not allowed to access the privileged resources (such as the IDT and page tables) and process the privileged events (interrupt and exceptions). It must switch to the untrusted code to gain access to privileged resources and handle the events. The I/O-intensive and memory-intensive tasks essentially involve the frequent world switches which are expensive and introduce non-negligible performance losses, even though both software and hardware optimizations have been proposed trying to reduce the context switch latencies [61, 66, 73]. In this section, we introduce the three enclave operation modes supported by HyperEnclave, as shown in Figure 5. The world switches in different enclave operation modes are shown in Figure 6.

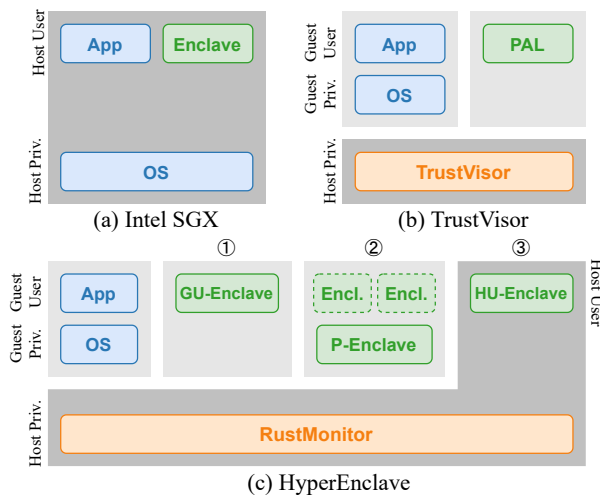


Figure 5: Comparison of the enclave operation modes supported by process-based TEEs. (a) Intel SGX runs enclaves in the host user mode (or guest user mode in the virtualization environment). (b) TrustVisor runs the protected code (Pieces of Application Logic, PALs) in the guest user mode. (c) HyperEnclave supports 3 coexisting enclave operation modes: ① GU-Enclaves running in guest user mode; ② P-Enclaves running in guest privileged mode and optional guest user mode; ③ HU-Enclaves running in host user mode.

4.1 Guest User Enclaves

Guest user enclave (GU-Enclave) is the basic enclave operation mode which is typically running computing-intensive tasks. The enclave runs in the guest user mode (i.e., guest ring-3 of the VMX non-root operation mode).

During the enclave creation, RustMonitor prepares a vCPU structure which contains a guest page table (GPT) and a nested page table (NPT) for GU-Enclave. On entry and exit between the normal VM and the enclave VM, RustMonitor switches the vCPU states (e.g. the instruction pointer, thread pointer, NPT, and GPT) accordingly.

To handle the interrupts and exceptions during the enclave running, RustMonitor configures the vCPU to trap all interrupts and exceptions to the monitor mode. RustMonitor then saves the enclave’s context, forwards the interrupt or exception to the normal VM. After the primary OS completes handling the interrupt or exception, the application invokes the ERESUME hypercall, which traps to RustMonitor to restore the enclave’s context and resume the execution of the enclave.

4.2 Host User Enclaves

Host user enclave (HU-Enclave) is running in host user mode. It delivers the optimal world switch efficiency by substituting the mode switch (hypercalls: ~ 880 CPU cycles on our platform) with the ring switch (syscalls: ~ 120 CPU cycles

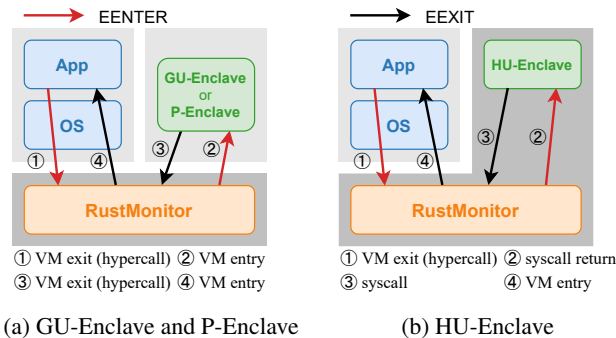


Figure 6: World switches for the supported enclave operation modes. (a) Using hypercalls to enter and exit GU-Enclave and P-Enclave. (b) Using syscalls and syscall returns to enter and exit HU-Enclave.

on our platform) (Figure 6). It further eliminates the extra virtualization overhead (e.g. vCPU context switching and two-dimensional page walking) in GU-Enclave. HU-Enclave may benefit the I/O-intensive workload according to our evaluation in Sec 7. By comparison, running enclaves in the guest user mode provides more defensive depth.

When loading the HU-Enclave, RustMonitor prepares a process context, e.g. creates a level-1 page table. On enclave entry, RustMonitor updates the CPU state and invokes the system call return instruction (i.e., SYSRET on x86 platforms) to enter the HU-Enclave. Correspondingly, on enclave exit, HU-enclave invokes the system call instruction (i.e., SYSCALL on x86 platforms) and traps into RustMonitor. The ENCLU leaf instructions (e.g., EGETKEY, EREPORT) are emulated as a system call. Interrupts and exceptions within the HU-Enclaves also trap into the RustMonitor. The procedures are similar to those for the GU-Enclaves described in Sec. 4.1.

4.3 Privileged Enclaves

Inspired by the VM-based TEEs, such as AMD SEV [45], HyperEnclave supports privilege enclaves (P-Enclaves) which run in guest privileged mode. P-Enclave is permitted to access the GDT, IDT, and level-1 page table which benefits a wide variety of applications, as demonstrated by Dune [21]. One such example is the garbage collector, an essential feature for Java applications (existing works port the JVM to enclaves [26, 43]). The garbage collector frequently changes page permissions to trigger page faults in order to track the page status. For user mode enclaves (e.g., GU-Enclaves and HU-Enclaves), it has to involve the primary OS to update the page table and handle the page fault which suffers huge performance loss due to world switches. P-Enclaves eliminate the world switch by supporting in-enclave exception handling and level-1 page table management. More specifically, P-Enclaves configures its own exception handler to handle certain exceptions (such as page fault). RustMonitor passes

through the white-list exceptions to the P-Enclave and forwards others to the primary OS. Furthermore, P-Enclaves can also support page-table-based in-enclave isolation schemes, e.g., sandboxing untrusted third-party libraries.

With the ability to receive interrupts within the enclaves, P-Enclaves may also detect abnormal interrupt events by counting the frequency, before requesting RustMonitor to route them to the primary OS. As such, existing interrupt-based side channel attacks [24, 37, 40, 58, 59, 70] could be detected and mitigated. We leave further exploration in this direction to future work due to space constraints.

5 Implementations

We report our implementation of HyperEnclave on an AMD platform that supports hardware virtualization technology and memory encryption. In the current implementation, RustMonitor consists of about 7,500 lines of code written mostly in Rust, and the kernel module for the primary OS has about 3,500 lines of C code. Also, we made about 2,000 lines of code changes to the official Intel SGX SDK (version 2.13).

5.1 RustMonitor

RustMonitor runs at the highest privilege level and enforces the isolation for the enclaves. To reduce the risks caused by memory corruption or concurrency bugs, we implemented RustMonitor mostly in Rust, a memory-safe language, with only a few lines of assembly code used for context switches. Compared with existing hypervisors such as KVM [47] and Xen [29], RustMonitor is much smaller and thus easier to be formally verified. We are working on the formal verification of RustMonitor and plan to release the result as a separate report.

When the platform is booted, we configure the kernel command line parameters in the grub to reserve regions of physical memory, which are exclusively used by RustMonitor and the enclaves. RustMonitor manages the reserved physical memory by maintaining a list of free pages. When an enclave page is needed, e.g., when adding an enclave page during enclave creation, a free page is retrieved from the pool; when the enclave page is freed, the page is attached to the list again. Moreover, RustMonitor also manages the enclave's page tables and processes the page fault.

5.2 The Kernel Module

The kernel module is loaded by the primary OS during the booting process. Then it loads, measures, and launches RustMonitor, with the measurement extended to the TPM PCR as part of the TPM quote. When the kernel module is loaded, a device file is created and mounted at `/dev/hyper_enclave`. The application can open it and issue the `ioctl()` to invoke the emulated privileged operations.

5.3 The Enclave SDK

HyperEnclave retrofits the official SGX SDK as follows.

Supporting the SGX SDK APIs. We replace the SGX user leaf functions (e.g. `EENTER`, `EEXIT`, `ERESUME`, etc.) in the SGX SDK with hypercalls or system calls. Our implementation retains the same parameter semantics and orders as SGX for compatibility purposes.

Parameters passing with the marshalling buffer. In HyperEnclave, the enclave can only access its own address space and the marshalling buffer shared with the application. The size of the marshalling buffer can be configured in the enclave's configuration file, with a default size. The data needs to be transmitted to the marshalling buffer before invoking edge calls. We modified SGX SDK to handle the transitions, which are thus transparent to the developer.

We modified the untrusted runtime library in the SDK (i.e., `libsgx_urts.so`), such that during enclave initialization a marshalling buffer is allocated using `mmap()` with `MAP_POPULATE` flags set. As a result, the GPAs for the marshalling buffers are pre-populated. Then an `ioctl()` is issued to request the primary OS not to compact or swap out the physical pages of the marshalling buffers during the enclave's lifetime. When the application invokes the emulated `EINIT` instruction to mark the initialization of the enclave, the base address and the size of the marshalling buffer are passed to RustMonitor, who will add the mapping of the marshalling buffer in the enclave's page table. In this way, the marshalling buffer is now shared between the enclave and the untrusted application. The base address and the size of the marshalling buffer are also passed to the trusted runtime library to transmit data from the marshalling buffer to the enclave.

The current `OCALL`'s implementation in the SGX SDK invokes the `sgx_ocalloc()` within the enclave to allocate a buffer on the stack area of the untrusted application, which is then used for cross-enclave data transmission. As such, we only need to modify the `sgx_ocalloc()` function to allocate a memory area in the marshalling buffer. To support parameter passing through the marshalling buffer for `ECALLs`, we modified SGX's `Edger8r` tool to automatically generate code that copies the transmitted data into the marshalling buffer.

The SGX programming model supports passing parameters with the `user_check` attribute. For such parameters, the SDK tool will not generate code to check the address range or perform data movement. Since the enclave code could access the entire process's address space, some enclave programs may use a pointer with the `user_check` attribute to manipulate the data buffer outside the enclave directly, without accounting for the overhead for copying the data across the enclave boundary. To deal with it, we added an interface for the developer to allocate the buffer within the marshalling buffer, in the cases when the developer may use parameters with the `user_check` attribute.

The remote attestation flow is similar to SGX, following

the same SIGn-and-MAc (SIGMA) protocol. We extended the `sgx_quote_t` structure in the SDK to include the HyperEnclave quote, and the modification is transparent to the enclave code.

With the above design, most SGX programs could run on HyperEnclave without source code changes. Furthermore, to ease the development of HyperEnclave applications, we have also ported the Rust SGX SDK [71] and the Occlum library OS [64] to HyperEnclave.

6 Security Analysis

Trust Establishment. HyperEnclave relies on measured boot to bootstrap the trust of RustMonitor, a common approach for the design of TEEs (e.g., TrustZone [16] and Keystone [49]). All the components during booting (including the CRTM, BIOS, grub, kernel, and initramfs) are measured and extended to the TPM PCRs. As a result, any tampering of the booted code will be reflected and audited through remote attestation.

We consider HyperEnclave is deployed in a controlled environment (i.e., the data center in the cloud computing scenario) such that the attacker has limited physical access to the platforms. To reduce the attack surface and minimize the TCB, we put the RustMonitor image into the initramfs, and RustMonitor is loaded and measured in early userspace. In this phase, the primary OS kernel does not accept external inputs from the user, and the peripherals such as network connection are disabled. With the measured late launch approach, RustMonitor does not need to trust the primary OS anymore after the OS is demoted to the normal mode.

Enclave memory isolation. As presented in Sec. 3.2, the enclave’s memory and page tables are maintained by RustMonitor, which are inaccessible to the primary OS. The TLBs are cleared upon world switches to prevent illegal memory accesses using stale TLB entries. RustMonitor prevents the primary OS to access the reserved physical memory by removing the corresponding mappings from its NPT. RustMonitor also configures the IOMMU to prevent unauthorized device accesses to the reserved physical memory.

Our design prevents the memory mapping attacks since the primary OS cannot interfere with the enclave’s address mappings. We introduce the marshalling buffer to support memory sharing between the enclave and the application. The application pre-allocates a marshalling buffer in normal memory and passes the base address and size of the buffer to RustMonitor during enclave initialization. In case the application may pass crafted addresses (e.g., to overwrite the enclave memory), before adding the mapping of the marshalling buffer to the enclave’s page table, RustMonitor ensures the address range of the marshalling buffer is outside the enclave address range.

Defense Against Compromised Enclaves. Previous work demonstrates that enclave malware may steal secret data or hijack the control-flow of the application outside the enclave [63]. HyperEnclave is designed to confine potentially

malicious enclaves as follows.

- *Preventing arbitrary memory accesses to the application.* The SGX enclave could access the entire address space of the application, and it is possible for attacks such as leaking the secret keys or stack canaries, or tampering with the code pointers for control flow attacks. In HyperEnclave, the enclave can only access its own memory and the marshalling buffer, which is only used for parameter passing.
- *Preventing arbitrary control flows after EEXIT.* The SGX design allows the enclave to jump to arbitrary addresses by setting `rbx` before executing the EEXIT instruction (i.e., exiting the enclave), opening door to enclave malware attacks [63]. In our design, since the EEXIT instruction is emulated by RustMonitor, it is easy to prevent such attacks by adding the validity check when EEXIT is invoked.

Physical attacks. Hardware memory encryption techniques such as AMD SME could be used to protect the enclave from physical attacks such as cold boot attacks or bus snooping attacks. With memory encryption, data are always encrypted in the memory or on the memory bus, and are only decrypted within the CPU. The memory encryption key is generated randomly on system boot and stored in the CPU, which cannot be accessed explicitly by software.

Side channel attacks. Compared with SGX, HyperEnclave can mitigate certain types of side channel attacks. Since the enclave’s guest page tables and page fault events are processed by RustMonitor without primary OS involvement, the latter cannot mount page-table-based attacks [25, 72, 74]. We leave the protection against micro-architectural attacks such as speculative execution attacks as future work.

7 Evaluation

We deployed HyperEnclave on a server with two AMD EPYC 7601 CPUs (2 threads per core, total of 128 logical cores) with 512 GB DDR4 RAM. We configure 2 GB reserved memory for RustMonitor, and 24 GB for EPC memory. The primary OS is Ubuntu 18.04 LTS with Linux kernel 4.19.91. For comparison, we run the same experiments on an Intel Xeon E3-1270 v6 CPU with SGX enabled, with 64 GB DDR4 RAM, running the same OS. The SGX SDK version for both HyperEnclave and the native SGX hardware are 2.13. All programs are compiled with GCC 7.5.0 and the same optimization level.

We tried to rule out differences between hardware. Except where explicitly stated, the evaluations didn’t exceed the EPC size, so as not to trigger excessive page swapping. All evaluations were performed in single-threaded mode. For micro-benchmarks (Table 1 and Table 2), we compare HyperEnclave on AMD hardware with SGX on Intel hardware using the same SGX SDK. We measured the core cycles to avoid the influence of CPU frequencies. For real-world workloads, we set the baselines as the counterparts with no security protections on Intel and AMD platforms respectively, and compare

	EENTER	EEXIT	ECALL	OCALL
Intel SGX	-	-	14,432	12,432
HU-Enclave	1,163	1,144	8,440	4,120
GU-Enclave	1,704	1,319	9,480	4,920
P-Enclave	1,649	1,401	9,700	5,260

Table 1: Latency of SGX primitives on HyperEnclave and Intel (in CPU cycles).

	Intel SGX	GU-Enclave	P-Enclave
#UD	28,561	17,490	258
#PF	-	2,660	1,132

Table 2: Average CPU cycles of handling an #UD and #PF exception inside the enclaves.

the *relative slowdowns* introduced by SGX and HyperEnclave. Since we only compare the relative slowdowns, we stress that the *absolute* performance results are on dissimilar platforms and are not directly comparable. All HyperEnclave evaluations were measured with memory encryption enabled.

We’ve been careful in ensuring HyperEnclave implements the TEE functionality correctly. Still, memory encryption between SGX1 and HyperEnclave is different, i.e., Merkel tree and AES-CTR versus AES-XTS (see Figure 11 in Sec. A.3 for the evaluation of memory encryption overhead), which may explain the improvement for memory-intensive workloads. Besides, world switches for HyperEnclave (especially, HU-enclave) are faster, which explains the improvement for I/O-intensive workloads.

7.1 World Switches Performance

We measured the latency of edge calls (i.e., ECALLs and OCALLs) on both HyperEnclave (under different enclave operation modes) and Intel SGX. The test code runs empty edge calls with no explicit parameters 1,000,000 times and takes the median value. We also measured the instruction-level latency for the emulated EENTER and EEXIT instructions on HyperEnclave. We were not able to measure the instruction-level latency on SGX since the `RDTSCP` instruction is not supported within the enclaves on our SGX platform.

The results are shown in Table 1. It shows that HU-Enclave has the optimal edge calls performance as it reduces a mode switch (~ 880 cycles) to a ring switch (~ 120 cycles) while P-Enclave is slower than GU-Enclave for it needs to switch more privileged states during the world switches. All of the results are comparable with Intel SGX.

7.2 Enclave Exception Handling

We used the undefined instruction exception (#UD) and the page fault exception (#PF) to evaluate the enclave exception handling performance. In the #UD benchmark, the test code

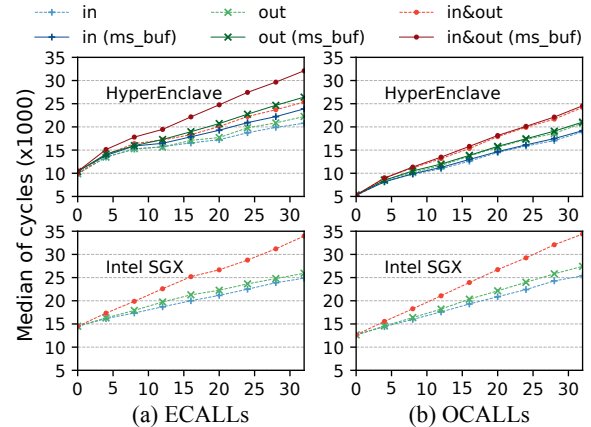


Figure 7: Marshalling buffer overhead for ECALLs and OCALLs with various data size, and with the marshalling buffer (marked as `ms_buf`) enabled and disabled.

executes an undefined instruction in the enclave to trigger the exception 1,000,000 times. The exception handler advances the instruction pointer and returns. For P-Enclave, the exceptions are captured and handled entirely within the enclaves, without enclave mode switches. For GU-Enclave and SGX, an exception causes an asynchronous enclave exit (AEX) and switches the CPU to the untrusted OS, then executes a two-phase exception handling [7]. The result shows that the exception handling within P-Enclaves is about $68\times$ and $110\times$ faster than GU-Enclave and Intel SGX respectively (Table 2).

We further simulated a typical garbage collector (GC) scenario that the test code first allocated a large memory buffer, then the write permissions to the buffer were revoked by changing the enclave’s page table. After that, the enclave accessed the buffer to trigger the page faults. In the exception handler, the write permission is restored. The result (Table 2) shows that P-Enclave is about $2.3\times$ faster than GU-Enclave, for P-Enclave updates the page table and handles the page faults by itself, while GU-Enclave needs to trap into RustMonitor to update the page tables. Note that we did not evaluate GC on Intel SGX, since our SGX1 platform does not support page permission modifications after the enclave initialization.

7.3 Marshalling Buffer Overhead

To measure the overhead of introducing the marshalling buffer, we constructed a GU-Enclave variant that does not use the marshalling buffer as the baseline. We measured the overhead for both ECALLs and OCALLs with various sizes of the transferred data while varying the directions for data movement (i.e., “in”, “out” and “in&out”). We ensure the data to be transferred is not cached using the `CLFLUSH` instruction. We evaluated the performance for transferring the same data on SGX for comparison.

Figure 7 provides the results for ECALLs and OCALLs respectively. It shows that the overhead increases almost lin-

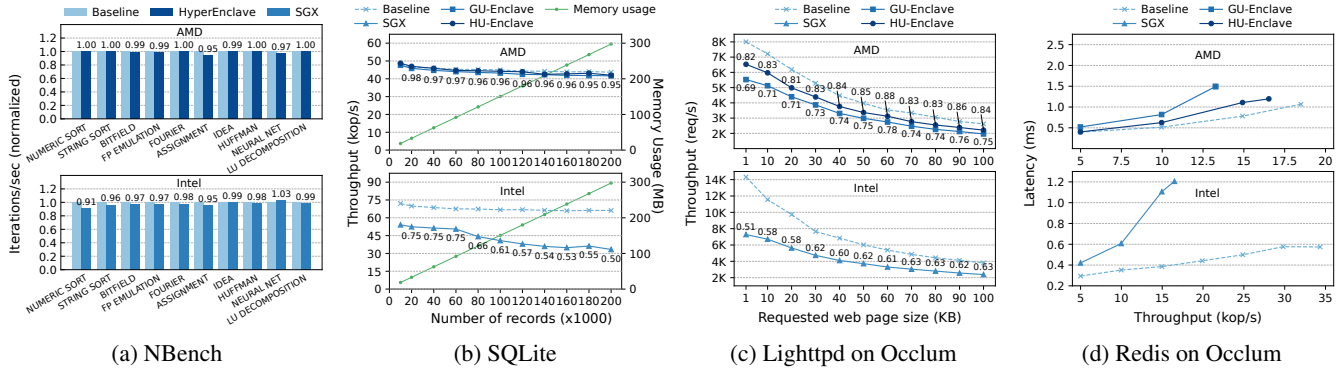


Figure 8: Performance of NBench, SQLite, Lighttpd, and Redis on AMD (with HyperEnclave) and Intel (with SGX).

early with the data size. For ECALLs, the overhead for “in”, “out” and “in&out” directions is 8%, 11% and 21% respectively for transferring 16 KB data, due to the extra memory copy. For OCALLs, the overhead is negligible, since it allocates a buffer on the marshalling buffer without additional memory copy (Sec. 5.3). We remind that data transfers in ECALLs contribute a small portion to the processing time for many real-world computation workloads, especially for computation or memory intensive tasks.

7.4 Real-world Workloads

The evaluations were conducted on four real-world applications: an algorithm benchmark suite **NBench** [53], a lightweight web server **Lighttpd** [13], two popular databases **SQLite** [14] and **Redis** [15], as representations for CPU intensive, I/O-intensive, and memory-intensive tasks. We ported the library OS Occlum [64] (v0.21) to the enclave SDK to reduce the porting effort for Lighttpd and Redis. We measured the performance on both HyperEnclave and Intel SGX, using the same code compiled under the SDK simulation mode as the baseline (providing no security guarantees).

NBench. NBench measures the performance of a system’s CPU, FPU, and memory system, without I/O and system calls involved. We used an adaptation of NBench to SGX, i.e., SGX-NBench [8] with no source code modification for our evaluation. As shown in Figure 8a, the overhead introduced by HyperEnclave and SGX is about 1% and 3% respectively.

SQLite. We ported SQLite (v3.19.3) with the enclave SDK, and evaluated it on both Intel SGX and HyperEnclave (GU-Enclave and HU-Enclave) using the YCSB [30] workload A (50% reads, 50% updates). In this evaluation we focused on the memory performance, so we configured the database as in-memory and embedded the client into the enclave to avoid I/O operations. We increased the number of records and measured the time for 100,000 database operations. As shown in Figure 8b, on SGX the throughput is about 75% of the baseline for small memory usage. When the memory usage exceeds the EPC size (about 90 MB), the performance drops to 50% due to page swapping. On HyperEnclave, both

GU-Enclave and HU-Enclave have almost the same performance as the baseline (< 5% overhead). We speculate that it’s because the memory encryption performance for AMD SME (without integrity protection) is faster than SGX.

Lighttpd. We ran a Lighttpd (v1.4.40) server with Occlum on both SGX and HyperEnclave (GU-Enclave and HU-Enclave modes). We used the Apache HTTP benchmarking tool [10] and ran 100 concurrent clients over the local loopback to fetch various sizes of web pages to evaluate the throughput. In this evaluation, the overhead mainly comes from the frequent enclave mode switches (Table 1). As shown in Figure 8c, HU-Enclave delivers the best performance as expected (81% ~ 88% of the baseline). GU-Enclave achieves 69% ~ 78% of the baseline, while SGX achieves 51% ~ 63% of the baseline.

Redis. We use Redis to evaluate the performance under the comprehensive scenarios where both memory and I/O are intensive. We ran a Redis (v6.0.9) database server with Occlum on both SGX and HyperEnclave (GU-Enclave and HU-Enclave modes). Similar to SQLite, we configured the database as in-memory and used the YCSB workload A. For the evaluation, we first loaded 50,000 records (in total 50 MB data) and then performed 100,000 operations from 20 clients over the local loopback. We increased the request frequency and measured the latency under different throughput. As shown in Figure 8d, HU-Enclave achieves 89% of the maximum throughput of the baseline, while GU-Enclave and SGX are about 72% and 48% of the baseline, respectively.

8 Discussions

HyperEnclave on other platforms. HyperEnclave requires the virtualization extension (specifically, two-level address translation) for isolation and TPM for the root of trust and randomness, etc. Virtualization is supported on many ARM servers (such as the ARMv8 platforms [2]). The RISC-V H-extension specification has evolved to v0.6.1 in 2021. Both ARM and RISC-V virtualization support two-level address translation. Certain TPM products already support ARM servers. Research has been conducted to support firmware TPM on RISC-V [22]. As such, signs are promising that

HyperEnclave can be adapted to run on ARM and RISC-V platforms.

However, porting HyperEnclave to ARM and RISC-V platforms requires non-trivial engineering effort, considering that the instruction set architectures (ISAs) are totally different. Take the ARMv8 architecture as an example. The software modules can be mapped to different exception levels (ELs): The monitor mode for RustMonitor can be mapped to EL2; The normal mode for the primary OS and untrusted part of the applications can be mapped to EL1 and EL0 respectively; The secure mode for enclaves can be mapped flexibly to EL1 or EL0. Memory isolation can be supported similarly with the support of stage 2 address translations. Furthermore, the official Intel SGX SDK only supports x86 platforms. In particular, the transitions across enclave boundaries are handled with platform-dependent assembly code, and need to be rewritten according to the application binary interface (ABI) of the targeted platforms. We leave the further exploration of adapting HyperEnclave to other platforms as future work.

Attack surfaces under different enclave operation modes.

The untrusted primary OS still runs within the VM and the attack surface from the primary OS to enclaves does not change. Running the enclaves in privileged mode or in the host, however, may expose more attack surfaces to a malicious enclave. For example, it may make the enclave malware easier to escalate to host ring-0, if the enclave runs in the host already.

9 Related Works

Most existing TEEs require specific hardware or firmware changes [17, 33, 39, 39, 41, 45, 55]. Specifically, CURE [20] changes the CPU core to support the enclave identifier (eid), and modifies the system bus to support the memory and peripheral arbiters for memory and peripheral access control according to the eid. Then the trusted security monitor can configure the hardware primitives to support the flexible enclaves. In contrast, HyperEnclave supports flexible enclave modes on commodity hardware without hardware or firmware changes.

A line of research has been conducted to build the isolated execution environment (e.g., PAL for TrustVisor and HAP for InkTag) using virtualization extensions, including TrustVisor [54], InkTag [38], Overshadow [28], AWS Nitro Enclaves [3], Microsoft Defender Credential Guard [5] and Hyper-V Shielded VMs [6]. TrustVisor makes the PAL page table pages read-only to prevent memory mapping attacks. The design introduces much overhead during PAL registration and switches from/to PALs. Even worse, it triggers many NPT violations in high memory pressures scenarios, due to the updates to the access and dirty bits of the PAL page tables, introducing huge overhead [52]. In Inktag, the HAP page tables are managed by the hypervisor. It allows the untrusted OS to request the hypervisor to update the HAP page tables. Both TrustVisor and InkTag are susceptible to page-table-based attacks [72, 74]. Furthermore, these designs usually contain a

large code base in the TCB, including device drivers, guest IO emulation, network and block device virtualization, etc. For example, AWS Nitro Enclaves [3] are constructed by the host KVM and Linux, and thus the host Linux kernel is always trusted. We made the design choice to minimize the TCB of RustMonitor, which performs basic CPU/memory virtualization and enclave management. The primary OS kernel only needs to be trusted during the boot process and is demoted after RustMonitor starts.

Komodo [35] implements an SGX-like enclave protection model in the TrustZone environment. Keystone [49] supports customizable TEEs on RISC-V platforms. Komodo enclaves run in secure user mode, while Keystone enclaves run in U-mode and S-mode. In comparison, HyperEnclave supports flexible enclave mode (guest ring-3, guest ring-0/ring-3, and host ring-3). Both Komodo and HyperEnclave use page-table-based memory isolation, while Keystone uses PMP (physical memory protection) for memory isolation.

Recently, ARM introduced the Realm Management Extension (RME) in the forthcoming ARMv9-A architecture [1]. The monitor (running at EL3) enforces physical memory isolation among the secure, non-secure, and Realm worlds. The trusted Realm Management Monitor (RMM), which executes at EL2 in Realm security state (R-EL2), isolates the Realms from each other through the stage 2 page tables. Similar to HyperEnclave, the RMM is much simpler than a typical hypervisor and relies on the non-secure hypervisor for device emulation etc. RME requires architectural extensions, while HyperEnclave does not. Moreover, HyperEnclave decouples its trust chain from the CPU as much as possible to construct an open and cross-platform TEE.

10 Conclusion

In this paper, we proposed HyperEnclave, an open TEE that can run on various platforms with minimum hardware requirements. It supports the process-based TEE model, and SGX programs can run on HyperEnclave with little or no code changes. Moreover, HyperEnclave supports the flexible enclave operation modes to fulfill various enclave workloads. We implemented HyperEnclave on commodity AMD servers and deployed the system internally for real-world computations. We are working on the formal verification of the implementation and plan to open source to the community.

Acknowledgments

We would like to thank our shepherd David Cock and the anonymous reviewers for their invaluable feedback. This work was supported in part by the National Key R&D Program of China (Grant No. 2020YFB1805402) and the National Natural Science Foundation of China (Grant No. 61802397 and No. U19A2060).

References

- [1] Arm Realm Management Extension (RME) System Architecture. <https://developer.arm.com/documentation/den0129/latest>.
- [2] Armv8 white paper. <https://community.arm.com/docs/DOC-10896>.
- [3] AWS Nitro Enclaves User Guide. <https://docs.aws.amazon.com/enclaves/latest/user/nitro-enclave.html>.
- [4] ISO/IEC, P.D.: 11889: Information technology – Security techniques – Trusted platform module.
- [5] Manage Windows Defender Credential Guard. <https://docs.microsoft.com/en-us/windows/security/identity-protection/credential-guard/credential-guard-manage>.
- [6] Microsoft Hyper-V Shielded VM. <https://www.techtarget.com/searchwindowsserver/definition/Microsoft-Hyper-V-Shielded-VM>.
- [7] SGX two phase exception handling. <https://github.com/MWShan/linux-sgx/blob/master/docs/DesignDocs/IntelSGXExceptionHandling-Linux.md>.
- [8] The nbench benchmark ported to SGX. <https://github.com/utds3lab/sgx-nbench>, 2017.
- [9] Google. Asylo. <https://asylo.dev/>, 2019.
- [10] ab - apache http server benchmarking tool. <https://httpd.apache.org/docs/2.4/programs/ab.html>, 2021.
- [11] Intel 64 and IA-32 architectures software developer’s manual, combined volumes:1,2A,2B,2C,3A,3B,3C and 3D. <https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf>, 2021. Order Number: 325462-075US, June 2021.
- [12] Intel SGX for Linux. <https://github.com/intel/linux-sgx>, 2021.
- [13] Lighttpd. <https://www.lighttpd.net>, 2021.
- [14] SQLite. <https://www.sqlite.org>, 2021.
- [15] Redis. <https://redis.io>, 2022.
- [16] T. Alves and D. Felton. Trustzone: Integrated hardware and software security. *white paper*, 2004.
- [17] ARM. Arm Confidential Compute Architecture. <https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture>, 2020.
- [18] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O’keeffe, Mark L Stillwell, et al. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.
- [19] Ahmed M. Azab, P. Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, G. Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.
- [20] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stempf. CURE: A security architecture with customizable and resilient enclaves. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.
- [21] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, October 2012. USENIX Association.
- [22] Marouene Boubakri, Fausto Chiatante, and Belhassen Zouari. Towards a firmware TPM on RISC-V. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 647–650. IEEE, 2021.
- [23] F. Brasser, D. Gens, P. Jauernig, A. R. Sadeghi, and E. Stempf. Sanctuary: Arming trustzone with user-space enclaves. In *Network and Distributed System Security Symposium*, 2019.
- [24] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostianen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:sgx cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.
- [25] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, August 2017. USENIX Association.

- [26] Chia che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca A. Popa, and Donald E. Porter. Civet: An efficient java partitioning framework for hardware enclaves. In *USENIX Security Symposium*, 2020.
- [27] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. *ACM SIGARCH Computer Architecture News*, 41(1):253–264, 2013.
- [28] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dvoskin, and Dan RK Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review*, 42(2):2–13, 2008.
- [29] David Chisnall. *The definitive guide to the xen hypervisor*. Pearson Education, 2008.
- [30] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.
- [31] Standard Performance Evaluation Corporation. SPEC CPU 2017. <https://www.spec.org/cpu2017/>, 2021.
- [32] Victor Costan and S. Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.
- [33] Victor Costan, Iliia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, August 2016. USENIX Association.
- [34] McKeen F., Alexandrovich I., Anati I., Caspi D., Johnson S., Leslie H. R., and Rozas C. Intel software guard extensions (intel sgx) support for dynamic memory management inside an enclave. *Hardware and Architectural Support for Security and Privacy*, 2016.
- [35] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *SOSP'17*, 2017.
- [36] Trusted Computing Group. TPM Library Specification 2.0, 2016.
- [37] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*, pages 299–312, 2017.
- [38] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: secure applications on an untrusted operating system. *ASPLOS ... proceedings. International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 253–264, 2013.
- [39] Guerny DH Hunt, Ramachandra Pai, Michael V Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A Goldman, et al. Confidential computing for openpower. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 294–310, 2021.
- [40] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level directional predictor based side-channel attack against sgx. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 321–347, 2020.
- [41] Intel. Intel Trust Domain Extensions. <https://software.intel.com/content/dam/develop/external/us/en/documents/tdxwhitepaper-v4.pdf>, 2020.
- [42] Intel. Intel Architecture Memory Encryption Technologies Specification. <https://software.intel.com/content/dam/develop/external/us/en/documents-tps/multi-key-total-memory-encryption-spec.pdf>, 2021.
- [43] Jianyu Jiang, Xusheng Chen, Tsz On Li, Cheng Wang, Tianxiang Shen, Shixiong Zhao, Heming Cui, Cho-Li Wang, and Fengwei Zhang. Uranus: Simple, efficient sgx programming and its applications. *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020.
- [44] David Kaplan. AMD x86 memory encryption technologies. Austin, TX, August 2016. USENIX Association.
- [45] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.
- [46] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. Coin attacks: On insecurity of enclave untrusted interfaces in sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 971–985, 2020.
- [47] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

- [48] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.
- [49] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.
- [50] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O’Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eysers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for intel SGX. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*, pages 285–298, 2017.
- [51] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1619, 2015.
- [52] Andrei Lutas, Daniel Ticle, and O. Creț. Hypervisor based memory introspection: Challenges, problems and limitations. In *ICISSP*, 2017.
- [53] Uwe F. Mayer. Linux/Unix nbench. <https://www.math.utah.edu/~mayer/linux/bmark.html>, 2017.
- [54] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. *2010 IEEE Symposium on Security and Privacy*, pages 143–158, 2010.
- [55] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. *Hasp, isca*, 10(1), 2013.
- [56] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference (USENIX ATC 96)*, San Diego, CA, January 1996. USENIX Association.
- [57] Jāmes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Twine: An embedded trusted runtime for webassembly. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 205–216. IEEE, 2021.
- [58] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017.
- [59] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. Copycat: Controlled instruction-level attacks on enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 469–486, 2020.
- [60] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 619–636, 2016.
- [61] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 238–253, 2017.
- [62] Mark Russinovich. Introducing Azure confidential computing. *Seattle, WA: Microsoft*, 2017.
- [63] M. Schwarz, S. Weiser, and D. Gruss. Practical enclave malware with intel sgx. In *DIMVA 2019*, pages 177–196, 2019.
- [64] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 955–970, 2020.
- [65] Shweta Shinde, Shengyi Wang, Pinghai Yuan, Aquinas Hobor, Abhik Roychoudhury, and Prateek Saxena. Besfs: A POSIX filesystem for enclaves with a mechanized safety proof. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 523–540, 2020.
- [66] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. Switchless calls made practical in Intel SGX. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, pages 22–27, 2018.
- [67] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*, pages 645–658, 2017.
- [68] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E Porter. Civet:

An efficient java partitioning framework for hardware enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 505–522, 2020.

- [69] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1741–1758, 2019.
- [70] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemo: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–195, 2018.
- [71] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards memory safe enclave programming with rust-sgx. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.
- [72] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434, 2017.
- [73] Ofir Weisse, Valeria Bertacco, and Todd Austin. Re-gaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 81–93. IEEE, 2017.
- [74] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.
- [75] Minhong Yun and Lin Zhong. Ginseng: Keeping secrets in registers when you distrust the operating system. In *NDSS*, 2019.
- [76] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. Sectee: A software-based approach to secure enclave architecture using TEE. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

A Supplementary Materials

A.1 Mapping Attacks

See Figure 9.

A.2 Virtualization Overhead

We’d like to evaluate the virtualization overhead on the normal VM. We ran SPEC CPU 2017 INTSpeed benchmarks [31], LMBench [56], and Linux kernel (v5.15) building when enabled and disabled RustMonitor. The result shows that the virtualization overhead is less than 1% in most benchmarks (see Figure 10 and Table 3). HyperEnclave avoids massive VM-exits by pass-through most devices to the normal VM and installs huge pages in NPT when possible to relieve the TLB pressure.

	LMBench (μ s)						Kernel Build (s)
	null call	fork	ctxsw	mmap	Page Fault	AF UNIX	
Native	0.1195	196.3	3.13	66,125	0.2433	5.73	1,410
Normal VM	0.1192	197.9	3.22	66,407	0.2461	5.69	1,417
Overhead	-0.25%	0.82%	2.88%	0.43%	1.15%	-0.70%	0.50%

Table 3: Virtualization overhead on LMBench (null syscall, fork, context switches among 16 processes with 64KB working set, mmap, page fault, and unix socket) and building the Linux kernel.

A.3 Memory Encryption Overhead

We evaluate the memory encryption overhead by measuring the memory access latency with and without encryption in sequential and random access patterns. The buffer size is varied from 16 KB to 256 MB. Figure 11 illustrates the result on HyperEnclave and SGX. When the buffer size is smaller than the LLC size (8 MB), the overhead on both platforms is negligible. When the buffer size is over the LLC size, the overhead for sequential accesses and random accesses can be over $2.4\times$ and $25\times$ respectively on HyperEnclave, while on SGX is $3\times$ and $30\times$ respectively. When the buffer size exceeds the EPC size (93 MB), the overhead for sequential accesses and random accesses is $45\times$ and $1000\times$ slow on SGX, due to the EPC page swapping, while on HyperEnclave the overhead is still less than $30\times$ since we reserve 24GB as enclave memory in our test.

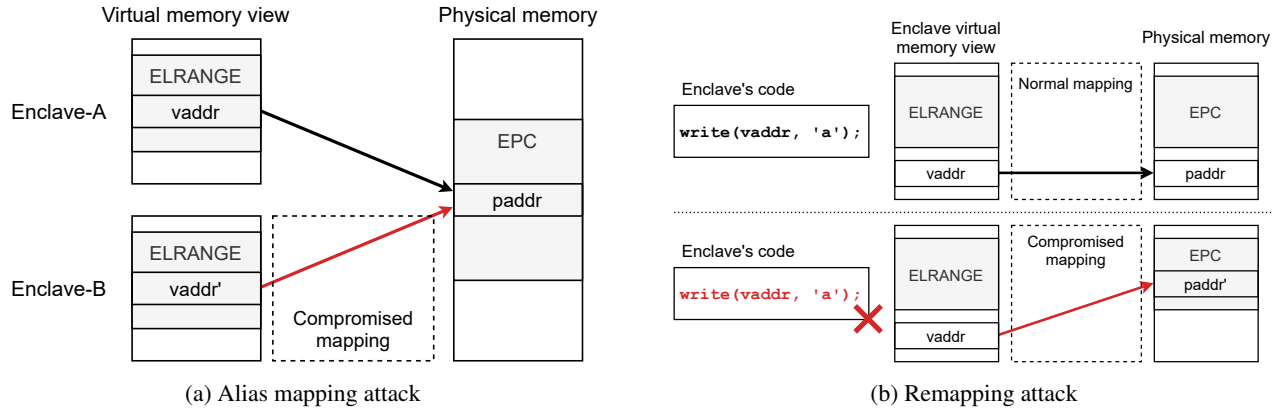


Figure 9: Mapping attacks. (a) Two guest virtual addresses within the enclaves are mapped to the same guest physical address; (b) A non-enclave virtual address is mapped to the physical address belonging to the enclave.

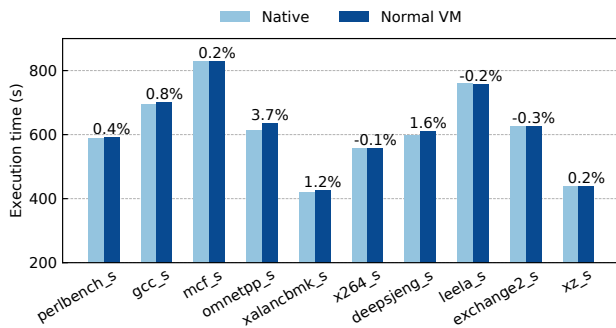


Figure 10: Virtualization overhead on SPEC CPU 2017.

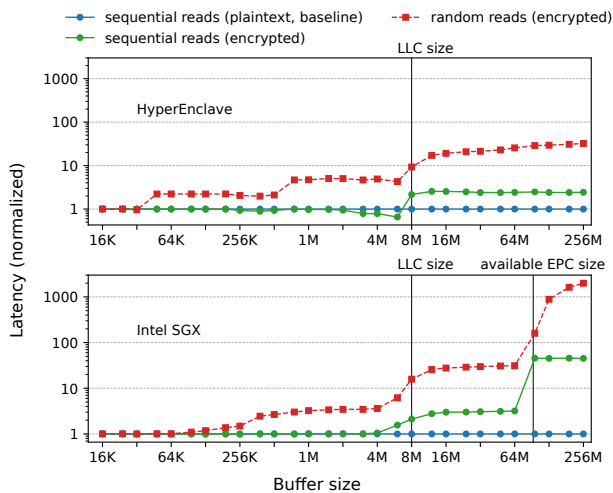


Figure 11: Memory encryption overhead for sequential and random memory accesses on HyperEnclave (with AMD SME) and Intel SGX (with Intel MEE). The LLC size is 8 MB, and the available EPC size on Intel SGX is about 93 MB.

B Artifact Appendix

Abstract

HyperEnclave can support existing SGX toolchains and run SGX applications on AMD CPU with security guarantees. This artifact contains the binaries of the RustMonitor, and documentations on how to setup the HyperEnclave environment. We provide two containers to reduce the environment configuration efforts. Specifically, the server container includes the pre-installed enclave SDK, the Occlum LibOS, and the benchmarks along with their dependencies⁵. The client container includes pre-installed client side benchmark scripts for Lighttpd and Redis.

Scope

The artifact includes benchmarks for edge calls (i.e., ECALLs and OCALLs), and benchmarks for the real-world workloads, including NBench, SQLite, Lighttpd and Redis. We provide scripts to reproduce the results in the paper (summarized in Table 4).

Contents

- `README.md` describes the artifact and provides a road map for evaluation.
- `host/` contains RustMonitor binary, the Linux kernel module binary, and the scripts to install and enable HyperEnclave.
- `server/` contains the source code (or patches) and scripts of all experiments to run within the enclaves. We also provide a docker container with all dependencies installed.

⁵The artifact is based on SGX SDK v2.15, Occlum LibOS v0.27, and GCC 9.4.0. The versions have little effect on the performance results.

Experiments	Figure/Table	Which container	Estimated time	Description
edge-calls	Table 1	server	10s	The latency of EENTER/EEXIT and ECALLs/OCALLs.
exception	Table 2	server	20s	Handling exceptions inside the enclaves.
NBench	Figure 8a	server	10m	Performance scores of NBench inside the enclaves.
SQLite	Figure 8b	server	15m	Throughput of in-memory SQLite database with different number of records, under YCSB A workload.
Lighttpd	Figure 8c	server/client	10m	Throughput of Lighttpd web server inside Occlum LibOS with different request sizes.
Redis	Figure 8d	server/client	20m	Latency-throughput curve of Redis in-memory database server inside Occlum LibOS with increasing request frequencies. The client uses YCSB A workload.

Table 4: Summary of the benchmarks included in the artifact.

- `client/` contains the benchmark scripts for network-based experiments (Lighttpd and Redis) to run on the client side. We also provide a docker container with all dependencies installed.
- `plots/` contains plotting scripts to generate figures from the experiment results.
- `paper-results/` contains the results shown in the paper.

Hosting

Check out <https://github.com/HyperEnclave/atc22-ae> (tag: `atc22-ae`, commit ID: `d1be8ab`).

Requirements

Hardware requirements:

- A 64-bit AMD platform with SVM enabled. Optionally, we recommend that the platform should support SME for the protection against physical memory attacks.
- RAM \geq 16 GB.
- Free disk space \geq 30 GB.

We disabled TPM and IOMMU features in RustMonitor binary for artifact evaluation to minimize the hardware requirements. These features do not affect the performance results.

Software requirements:

- Linux with the specified kernel version (i.e., `5.3.0-28-generic`) to match our given kernel module binary. We recommend Ubuntu 18.04.4 LTS which uses this version of kernel as the default.
- Docker.
- Git.
- GCC and Linux kernel headers (for building the `enable_rdfsbase` kernel module).

PRIDWEN: Universally Hardening SGX Programs via Load-Time Synthesis

Fan Sang^{*,1}, Ming-Wei Shih^{*,3}, Sangho Lee⁴, Xiaokuan Zhang¹,
Michael Steiner², Mona Vij² and Taesoo Kim¹

¹Georgia Institute of Technology, ²Intel Labs, ³Microsoft, ⁴Microsoft Research

Abstract

A growing class of threats to Intel Software Guard Extensions (SGX) is Side-Channel Attacks (SCAs). As a response, numerous countermeasures have been proposed. However, it is hard to incorporate them to protect SGX programs against multiple SCAs simultaneously. A naïve combination of distinct countermeasures does not work in practice because some of them are 1) *undeployable* in target environments lacking dependent hardware features, 2) *redundant* if there are already defenses with similar functionalities, and 3) *incompatible* with each other by design or implementation. Identifying all of such conditions and preparing potential workarounds before deployment are challenging, primarily when an SGX program targets multiple platforms that abstract or manipulate their configurations.

PRIDWEN is a framework that selectively applies essential SCA countermeasures when loading an SGX program based on the configurations of the target execution platform. PRIDWEN allows a developer to deploy a program in the form of WebAssembly (Wasm). Upon receiving a Wasm binary, PRIDWEN probes the current hardware configuration, synthesizes a program (i.e., a native binary) with an optimal set of countermeasures, and validates the final binary. PRIDWEN supports both software-only and hardware-assisted countermeasures, and our evaluations show PRIDWEN efficiently, faithfully synthesizes multiple benchmark programs and real-world applications while securing them against multiple SCAs.

1 Introduction

Conducting *confidential or private computing* in a shared computing environment (e.g., the public cloud) is challenging [1, 2]. Intel Software Guard Extensions (SGX) [3, 4] has thus been proposed and adopted by leading cloud service providers to help ensure even system software and hardware cannot compromise the authenticity, confidentiality, and integrity of applications running inside SGX *enclaves* [5–7]. Nevertheless, Intel SGX is susceptible to Side-Channel Attacks (SCAs) [8], which are threats to shared cloud environments in which it aims to be deployed. Researchers

*The two lead authors contributed equally to this work.

Attack	Known Countermeasures
Cache	Cache flushing [21], cache eviction detection [23]
Page	Page fault detection [24], frequent exception monitoring [26, 27]
HT	HT disabling [21], co-location detection [25, 26]
Branch prediction	Branch obfuscation [28]
Speculation	Branch prediction control [29], lfence [30]
L1TF	Cache flushing and HT disabling [21]
MDS	HT disabling [22]

Table 1: Known side-channel attacks against SGX and countermeasures. HT: Hyper-Threading; L1TF: L1 Terminal Fault; MDS: Microarchitectural Data Sampling.

have shown that SGX is vulnerable to various SCAs utilizing cache [9–13], page table [14–18], and transient execution [19, 20], which can infer sensitive control flows or exfiltrate secret data. To defend against individual SCAs, software- and/or hardware-based countermeasures have been proposed, such as cache flushing [21, 22] or eviction detection [23], page fault detection [24], and Hyper-Threading Technology (HT) disabling [21, 22] or co-location detection [25, 26] (Table 1).

However, multiple side channels can co-exist in a vulnerable program; protecting SGX programs from multiple known SCAs is difficult, not to mention the existence of unknown ones. One way is collectively applying existing countermeasures against individual SCAs, but naïvely doing so fails due to the unawareness of diverse target execution platforms or co-existing mitigation techniques, which may make such countermeasures 1) *undeployable* due to different hardware settings, 2) *redundant* because of over-protection, and 3) *incompatible* due to conflicts among different mitigations. Another way is adopting a comprehensive countermeasure, i.e., oblivious execution [18, 31], that is effective to many SCAs except for speculative execution. However, even the state-of-the-art oblivious execution incurs average slowdown of $51\times$ [31], largely downgrading the cost-effectiveness of cloud computing. A practical alternative, data-location (re-)randomization [32], incurs relatively small slowdown ($8\times$), but it is still heavy and does not cover control-flow leakage.

One conventional approach to solve such issues is to create a *bloated* application incorporating independently compiled object files for each architecture and runtime detection code, to selectively activate them according to different hardware configurations [33]. This approach, however, is not suitable for Intel SGX: First, checking hardware configurations is sup-

ported by system software outside the Trusted Computing Base (TCB); malicious system software can provide misinformation about hardware configurations to SGX applications. Second, the secure memory that enclaves share, Processor Reserved Memory (PRM), is limited [4]; bloated SGX applications can easily occupy a huge portion of it.

PRIDWEN. To practically protect SGX programs from various SCAs, we argue that the decision of the SCA mitigations to be applied should be delayed as close to the final execution as possible to best fit the target SGX platform as well as co-existing mitigation techniques. Therefore, instead of adopting the static compilation approach, in this paper, we propose PRIDWEN, a framework that uses *load-time synthesis* to dynamically harden SGX programs by selectively applying different mitigation techniques according to the configurations on the target execution platform. While ensuring the security, PRIDWEN maintains the cost-effectiveness of cloud computing by minimizing the extra effort required for preparation on the tenant side, and the runtime overhead of program synthesis on the cloud side.

PRIDWEN has a universal loader that securely loads and hardens a given SGX program inside an enclave based on four components: 1) Prober that identifies the target platform’s hardware and system configurations using SGX exception handling logic and remote attestation; 2) PassManager that manages and determines an optimal set of feasible instrumentation passes based on the identified configurations; 3) Synthesizer that hardens a given SGX program with the chosen instrumentation passes before loading it in the target enclave; and 4) Validator that verifies whether the final executable is hardened as expected, and provides a functionality for developers to remotely verify both the process of synthesis and the hardened binary before execution.

To make PRIDWEN 1) *platform-independent*, 2) *instrumentation-friendly*, and 3) *lightweight*, we develop a new instrumentation framework using WebAssembly (Wasm) [34, 35] as the Intermediate Representation (IR). The size of PRIDWEN in binary is only 1.26 MiB, which only adds a slim footprint to the enclave TCB. Existing Wasm runtimes for SGX [36–39] only *interpret* Wasm binaries without any *instrumentation support*. Furthermore, unlike existing Wasm instrumentation frameworks for non-SGX programs [40, 41], PRIDWEN can comprehensively transform Wasm binaries at both Wasm IR and native code levels. PRIDWEN also supports multiple high-level languages; users only need to compile their SGX programs once with a Wasm compiler backend (e.g., Emscripten [42]).

To showcase the capability and practicality of PRIDWEN, we integrate four mitigation passes into PRIDWEN: 1) T-SGX [24] to prevent a page-fault SCA with a hardware support; 2) Varys [26] to mitigate cache-timing, page-fault-, and HT-based attacks in a software-only manner; 3) QSpec-tre [30] to mitigate the Spectre attack; and 4) fine-grained Address Space Layout Randomization (ASLR) [43] as a

general-purpose mitigation. We first detail the steps to integrate the four passes into PRIDWEN, then we demonstrate how PRIDWEN produces the optimal set of passes based on the runtime configurations with minimal manual effort.

Performance. PRIDWEN poses moderate performance overhead on top of the original mitigation techniques and retains faithfulness of execution semantics (§6). The average slowdown of hardened real-world applications (Lighttpd, libjpeg, and SQLite) was $2.1\times$ with hardware-assisted non-redundant mitigation techniques and $3.6\times$ with software-only mitigation techniques for outdated microcode (i.e., no hardware-level mitigation), which closely conforms to the originally reported performance overhead of the selected countermeasures. Also, PRIDWEN faithfully compiled and ran all 73 programs from the official Wasm specification test suite [44]. Program synthesis completed within 0.5 s with a temporary usage of less than 25 MiB of enclave memory across tests.

PRIDWEN is designed to be an easily-extensible universal framework that respects the diversity of computing platforms. PRIDWEN is publicly available as an open-source project ¹, allowing communities to test, use, and contribute. We envision that the growing PRIDWEN framework should serve as a hub for the SCA-resistant SGX ecosystem, and potentially other mitigations as well.

Contributions. This paper makes the following contributions:

- **The first platform-aware load-time synthesis framework for SGX programs.** To the best of our knowledge, PRIDWEN is the first framework that dynamically synthesizes and hardens SGX programs by applying optimal hardware- and/or software-based mitigations according to the target platform.
- **Attestable in-enclave Wasm instrumentation and compilation toolchain.** A comprehensive instrumentation and compilation toolchain based on Wasm is implemented inside the SGX enclave to enable dynamic program synthesis with attestation. PRIDWEN can instrument Wasm both at IR and native level.
- **Extensive evaluation.** We study the performance of PRIDWEN extensively using benchmarks and real-world applications. The results suggest that PRIDWEN only introduces moderate runtime overhead while preserving the execution semantics.

2 Background and Related Work

In this section, we present the background and related work. Additional related work can be found at [Appendix A](#).

Intel SGX. Intel SGX is a hardware-based Trusted Execution Environment (TEE) that securely runs a userspace application in an untrusted remote environment, such as the public cloud. Through remote attestation [45], Intel SGX allows a user to

¹<https://github.com/sslab-gatech/Pridwen>

load his/her signed program into a remote environment, while helping ensure that the program binary has never been altered. To help secure the code and data of SGX programs, Intel SGX provides an enclave to each program, which is a dedicated secure region of the main memory. The enclave is isolated from any other software including an OS. The code and data stored in the enclave are always encrypted by the Memory Encryption Engine (MEE), and decrypted only when they are loaded into a CPU package (i.e., the cache), to help prevent physical attacks such as a cold boot attack [46].

Exceptions in SGX. Conventionally, exceptions are delivered to system software such as OS for further investigation. However, Intel SGX cannot adopt traditional exception handling because system software is untrusted. Instead, Intel SGX defines two mechanisms, Asynchronous Enclave Exit (AEX) and Custom Exception Handler (CEH) [47, 48]. Whenever an exception is generated during an enclave execution, the CPU exits from the enclave asynchronously. During AEX, the original exception number and register context are stored into the State Save Area (SSA) inside the enclave and then overwritten by synthetic data. Further, SGX allows developers to define CEHs to process exceptions inside an enclave; These CEHs can retrieve the SSA to check the stored exception number (`GPRSGX.EXITINFO.VECTOR`) and registers (`GPRSGX.<registers>`), and update them (e.g., `GPRSGX.RIP`) to resume the execution.

SGX side channels. SCAs against SGX have been actively studied. Traditional cache SCAs work against Intel SGX with different configurations [9–13, 49]. Recent studies show that page access patterns [14–18], interrupt execution time [50, 51], branch prediction behaviors [28, 52], speculative execution [19, 20] and Intel’s internal buffers [53–55] can all be used to infer sensitive information inside enclaves. In response, countermeasures that cope with the fundamental characteristics of the SCAs have been proposed. T-SGX [24] and Cloak [23] use Transactional Synchronization Extensions (TSX) to accurately recognize page faults and cache evictions inside an enclave, respectively. Varys [26] and Déjà Vu [27] aim to detect frequent interrupts or AEXs. Also, since HT enables concurrent SCAs without interrupts, Varys [26] and HyperRace [25] try to prevent an SGX hyperthread from being co-located with other hyperthreads. Disabling HT and/or flushing the L1 cache are also necessary to mitigate recent speculative or transient SCAs [21, 22]. In addition, SGX-LAPD [56] leverages a huge page to degrade the accuracy of the page-level SCA. Lastly, oblivious code execution and data access techniques for Intel SGX [18, 31, 57–60] have been proposed as a general countermeasure against SCAs, but they incur overly high performance overhead. The state-of-the-art ORAM-based system Klotski [61] improves the performance significantly. However, it only defeats controlled-channel attacks [14, 18]. In contrast, PRIDWEN focuses on universally hardening SGX applications by automatically and selectively applying multiple defenses against different SCAs together.

WebAssembly (Wasm). The World Wide Web Consortium (W3C) proposes Wasm [34] as a platform-independent compilation target for various high-level languages (e.g., C/C++ and Rust). A Wasm binary has language-like syntax and structure that are suitable for compilation and instrumentation. The basic executable unit of code in Wasm is a module that consists of multiple sections, where each section contains specific definitions of the module such as global variables, functions and a sequence of instructions of each function. Wasm instructions execute on a stack machine, and Wasm supports only the structured control flow such as `if-else` and `loop` without `goto` statements, enabling single-pass fast compilation.

Memory safety in Wasm. Wasm maintains a linear memory with a configurable size dedicated to all the memory accesses except for local and global variables. The linear memory is disjoint from other memory regions such as the code section and the call stack. As a result, given a buggy Wasm program (e.g., originating from a C program with a memory corruption bug), an attacker can only interfere with the data in the linear memory, but cannot tamper with its control flow.

PRIDWEN and Wasm. Because the Wasm binary is well-structured and friendly for efficient Just-In-Time (JIT) compilation, PRIDWEN adopts Wasm as IR for supporting load-time synthesis. PRIDWEN also benefits from Wasm’s memory-safety feature, mitigating code-reuse attacks against SGX [62, 63]. Moreover, the minimal footprint of Wasm fits PRIDWEN’s demand for a compact yet flexible instrumentation and compilation toolchain inside an enclave. Although existing compilers (e.g., LLVM) can fit into enclaves with extended size in new scalable CPUs [64], the TCB size will largely increase, and the migration effort would be significant as well.

3 Overview

Scenario. In this paper, we consider the widely-used *confidential-computing* scenario on the cloud, where the user wants to utilize the Intel SGX on the cloud to protect his/her data and applications. In this scenario, there are two entities: the cloud and the user. The user runs his/her applications inside enclaves, and wants to protect his/her data against side-channel attacks using PRIDWEN.

Threat model. Our threat model is similar to the threat models in other SGX-related studies [14, 24, 26]. Our TCB consists of an SGX enclave provided by an Intel CPU and everything inside the enclave, including PRIDWEN, a target Wasm binary prepared by the user, and the PRIDWEN pass selection policy. We assume that the user uses remote attestation to verify the validity of the CPU and PRIDWEN, and establish a secure channel with PRIDWEN to securely transmit his/her binaries. We assume that adversaries have already compromised the underlying privileged system software (e.g., OS) to attack PRIDWEN and the target binary. Any threats due to poten-

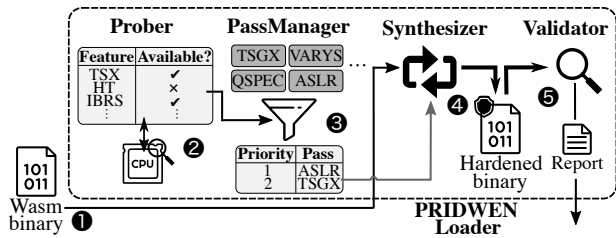


Figure 1: Overview of PRIDWEN. ❶ A user compiles a program into a Wasm binary and transmits it to PRIDWEN via a secure channel. ❷ PRIDWEN probes the hardware configurations. In this example, the CPU enables TSX and IBRS while disabling HT. ❸ PRIDWEN selects mitigation passes. Here, it chooses T-SGX and ASLR because the CPU enables TSX (for mitigating page-fault attacks) and IBRS (for mitigating Spectre variants). ❹ PRIDWEN synthesizes and hardens a native binary based on chosen passes. ❺ PRIDWEN validates the final synthesized native binary. A report is sent back to the user to attest the final binary.

tial vulnerabilities of the CPU and the code running inside enclaves are out of scope.

Goals. PRIDWEN is designed to achieve the following goals:

1) *Adaptivity.* PRIDWEN selects mitigation techniques that conform to the capabilities of the target execution platform on demand. PRIDWEN needs to optimally combine multiple mitigation techniques without causing conflicts or failures.

2) *Attestability.* The second goal of PRIDWEN is to support the remote attestation of the dynamically generated binary inside SGX; Native SGX only supports attesting static binaries. PRIDWEN should allow users to verify the integrity of the final executable running inside an enclave, as well as obtain the genuine information regarding whether the executable is faithfully generated by PRIDWEN (e.g., the selection and application of the mitigation schemes).

3) *Extensibility.* Another goal of PRIDWEN is to be extensible, so that it can support forthcoming mitigation techniques against SCAs besides existing ones. Moreover, it should support multiple platforms due to the diversity of practical computing platforms. The extensibility of PRIDWEN should also allow for smooth integration of legacy mitigation techniques.

Architecture. Figure 1 shows an overview of PRIDWEN. The core of PRIDWEN is an in-enclave loader that implements key ideas with corresponding components: *user-mode hardware probing* (Prober), *optimal pass selection* (PassManager), *load-time program synthesis* (Synthesizer), and *post-synthesis validation & final binary attestation* (Validator). Given that each countermeasure may depend on specific hardware features, Prober interacts with the platform and dynamically determines the availability of these features. Based on the probing results, PassManager determines an optimal set of countermeasures (i.e., instrumentation passes) and finalizes the order of applying them based on user policies. Next, PassManager informs Synthesizer about the final selection. Synthesizer takes a Wasm binary (provided by the user via a secure network channel) as an input and compiles it into a native one. During the com-

```

1 #define UD 6 /* Invalid opcode exception */
2 bool tsx_support = false;
3 check_tsx_support:
4   _xbegin();
5   tsx_support = true;
6   _xend();
7 exception_handler:
8   if (SSA.GPRSGX.EXITINFO.VECTOR == UD &&
9       SSA.GPRSGX.RIP == check_tsx_support) {
10    GPRSGX.RIP = skip_tsx_check;
11  }

```

Figure 2: Exception-based probing code for TSX. If a CPU does not support TSX, there will be a #UD exception that needs to be handled by an in-enclave exception handler to proceed execution (i.e., changing GPRSGX.RIP).

pilation, Synthesizer hardens the binary with the optimal pass set provided by PassManager. Validator takes the synthesized binary as an input, and verifies that 1) each countermeasure is correctly enforced, and 2) no conflict exists among the enforced countermeasures. Validator also provides the functionality of attestation on the final binary.

4 PRIDWEN

4.1 Prober

The goal of Prober is to identify hardware capabilities of the target execution platform, which is needed by PassManager to determine the optimal set of mitigation schemes to enforce. This *hardware probing* step typically requires interactions with the system software, such as retrieving privileged registers (i.e., Model-Specific Register (MSR) and control registers) and executing the cpuid instruction. However, we cannot rely on these approaches because the system software is not trusted in our threat model. SGX provides an attribute field called XSAVE-Feature Request Mask (XFRM) to determine whether some hardware features are enabled at enclave creation, but it only covers a few instructions (e.g., AVX and MPX [48]). To solve this, PRIDWEN leverages exception handling and remote attestation to securely probe hardware configurations while running inside an enclave.

Exception-based instruction probing. The instruction probing identifies whether PRIDWEN can use hardware-assisted mitigation techniques relying on specific instructions. A CEH for SGX (§2) can be used to determine whether a target system supports or enables the required instruction. Specifically, the probing code executes the specific instruction demanded (e.g., TSX) inside SGX, and then checks whether it results in a #UD exception by inspecting the exception information (Figure 2). If it does—i.e., the target platform does not support the instruction, PRIDWEN adopts a software replacement of the hardware-assisted mitigation if available, or omits it otherwise. The CEH then advances GPRSGX.RIP to continue execution.

Attackers can disrupt this type of probing, but it only results in Denial-of-Service (DoS) that they can always trigger without special attacks: 1) Attackers can simply resume the

API	Hooking point
onFunctionStart(CCTX *c)	Beginning of a function
onFunctionEnd(CCTX *c)	End of a function
onControlStart(CCTX *c)	Beginning of a control statement
onControlEnd(CCTX *c)	End of a control statement
onInstrStart(CCTX *c)	Before a IR-level instruction
onInstrEnd(CCTX *c)	After a IR-level instruction
onMachineInstrStart(CCTX *c, MI *i)	Before a native instruction
onMachineInstrEnd(CCTX *c, MI *i)	After a native instruction

Table 2: The APIs for the instrumentation. CCTX: CompilerContext. MI: MachineInstr. MCTX: MachineContext. MB: MachineBasicBlock.

enclave execution right after the #UD exception without invoking the CEH. However, GPRSGX.RIP still points to the invalid instruction, and it is impossible to manipulate it or the exception number outside the enclave. Therefore, this only incurs repeated #UD exceptions. 2) Attackers can selectively enable a specific hardware instruction *only during probing* and disable it during the actual execution. This trick can deceive the probing, but it only introduces #UD exceptions during runtime, resulting in another DoS.

Remote attestation for hardware configuration. The remote attestation determines whether a target platform is vulnerable to SCAs that utilize certain hardware features. SGX remote attestation allows PRIDWEN to accurately determine several hardware configurations, i.e., HT and Indirect Branch Restricted Speculation (IBRS). If a remote device turns on HT, an attestation verification report will contain CONFIGURATION_NEEDED in the `isvEnclaveQuoteStatus` field since API version 3 [65, 66]. PRIDWEN can leverage this information to selectively adopt mitigations for preventing hyperthread co-locations [25, 26]. Also, if a remote device does not install the microcode update for indirect branch control mechanisms, a remote attestation protocol will indicate `GROUP_OUT_OF_DATE` [67]. If users still want to securely run their code on such an outdated device, they can adopt software-based approaches [30] against speculative SCAs. Without learning such hardware information, unnecessary performance overhead might be paid for applying redundant protections. It is worth mentioning that updating microcode and changing HT configuration require system reboot in general; As a result, malicious system software cannot manipulate these hardware configurations during the execution of hardened programs.

4.2 PassManager

PassManager is in charge of selecting and integrating multiple mitigation schemes. PassManager provides a set of high-level APIs that allows developers of side-channel mitigation schemes to implement their instrumentation passes and plug them into the PRIDWEN loader. During the load time, PassManager 1) maintains a list of plugged-in passes, 2) determines the optimal set of passes for Synthesizer to execute, and 3) resolves the correct application order of each selected pass to avoid conflicts.

Pass APIs. Table 2 lists the high-level APIs for implementing instrumentation passes. For instrumentation, we expose all the hooks as APIs. To reflect the structure of a Wasm module, we classify the IR-level hooks into the granularity of functions, controls, and instructions. Each hook can obtain the information about the hooking IR instruction and the current states of compilation via the `CompilerContext` (CCTX) data structure. For the native level, the hook should consult the information of the native instruction via the `MachineInstr` (MI) data structure, as the `CompilerContext` data structure does not track such information.

Pass selection and ordering. When being plugged into the PRIDWEN loader, each pass is associated with a configuration file that specifies the type of SCA to mitigate, hardware features or other passes that it depends on, and a list of passes incompatible with it. Each pass can also specify its weakly dependent passes, which indicates that the pass depends on these weakly-dependent passes only when they are available. During the initialization phase, PassManager adds all the plugged-in passes into a pass queue. For better flexibility, PRIDWEN also allows a user to customize the pass queue by providing a pass selection policy (P), which contains descriptions of all plugged-in passes and their dependencies. We detail our implementation of a pass selection policy in §5.2.

To select the optimal set of passes, PassManager takes the following steps: 1) It consults Prober about the current hardware configuration. 2) It checks the dependency of each pass in the queue, and drops a pass if the required hardware feature is not available. 3) It checks the types of side channels that each active pass mitigates; if PassManager identifies more than one passes targeting the same SCA, it retains the one with the highest priority value specified in P . If not specified, it will first assign a priority value to each of them based on several criteria including performance overhead. 4) To determine the application procedure of active passes, PassManager builds a dependency graph of all the passes given the dependencies specified in P . Next, PassManager uses the topological order of the graph as the application order. PassManager may drop passes if their strong dependencies are not satisfied or incompatible passes are in the active pass set². If all passes are independent, PassManager uses the order in the pass queue as the application order. Here we assume the graph contains no circular dependencies; otherwise, PRIDWEN will terminate the execution.

4.3 Synthesizer

Synthesizer uses *load-time synthesis* to dynamically generate a final binary hardened with the optimal set of mitigation passes (§4.2) for the current hardware configuration, and loads

²Note that we did not come across any SCA mitigations that are mutually exclusive. If such cases emerge in the future, programmers can specify this situation in the pass selection policy (P) and mark the involved mitigation passes as mutually exclusive; the pass with higher priority will be applied by default. Users may override the policy to choose a custom priority.

the binary into memory for execution. Synthesizer adopts a Wasm binary as the input, and takes three steps, i.e., parsing, compilation, and instrumentation, to achieve this goal. We extend the compilation chain to support both IR- and native-level instrumentation so that it is flexible enough to integrate various types of SCA mitigation schemes with PRIDWEN.

Parsing. In the parsing step, Synthesizer performs standard decoding on a Wasm binary and converts it into Wasm IR. During decoding, Synthesizer also validates the format of the binary with several checks (e.g., type checking of functions) to guarantee that the binary follows the specification. Any modification to the binary before parsing can thus easily result in an immediate rejection. For example, inserting an instruction that causes the inconsistency on the stack machine renders the binary invalid.

Compilation. To generate the native binary inside the enclave given Wasm IR, Synthesizer performs a single-pass compilation over each function (similar to the baseline compilation of SpiderMonkey [34] and V8 [68]). During the compilation of a function, Synthesizer virtually executes each instruction based on the execution model of the Wasm stack machine and generates the corresponding native code. Synthesizer also keeps track of the metadata about each value (e.g., actual location and data type) on the operand stack to help correctly generate the native code and facilitate type-checking. In addition to the operand stack, Synthesizer maintains a control stack that keeps track of the control flow of the function. Pushing a value to the control stack indicates the function initiates a new control statement (e.g., block, if, or loop instruction), while popping a value from the control stack implies reaching the end of the current statement (e.g., an end instruction). The control stack provides sufficient information for Synthesizer to resolve the target of a branch (e.g., a br instruction). After finishing the native code generation of all functions, Synthesizer performs relocation. This process patches all the unresolved address values in the native instructions, such as call and those for memory accesses.

Instrumentation. To support flexible instrumentation, we extend the design of the compilation to provide hooks at both IR- and native-level. For IR-level hooks, we place them both before and after the position that Synthesizer processes an IR instruction to support code insertion, modification, and deletion. For each hook, we provide sufficient information about the corresponding instruction and the states of the compilation at the given point, such as the operands and the control stacks. Since Synthesizer may generate more than one native instruction for a single IR instruction, we provide similar hooks at the native level (i.e., surrounding the generation of native instructions) to support mitigation schemes that require the information about native instructions. To support the insertion or the modification of native instructions that require relocation, we provide the option to mark such instructions with symbols. A symbol refers to a target location that allows

Synthesizer to recognize and resolve it during the relocation phase.

Reproducible synthesis. Since both the compilation and instrumentation are *deterministic*, Synthesizer has a nice property: the synthesis process is reproducible. This property ensures that given the same Wasm code and hardware configuration, the same version of PRIDWEN loader always generates the same final binary.

4.4 Validator

The flexibility of instrumentation indicates that an instrumentation pass can arbitrarily modify the binary. Such modifications can potentially disturb the already applied instrumentation passes or break the binary itself. To avoid such cases, Validator supports *post-synthesis validation* to validate whether the synthesized executable is hardened as expected. Also, since the runtime behavior of PRIDWEN cannot be determined beforehand, Validator provides *final binary attestation* to allow users to remotely verify both the process of synthesis and the hardened binary before execution. Both *post-synthesis validation* and *final binary attestation* are necessary to ensure the correctness of the final binary. In addition, they are conducted *only* once before running the program, and thus will not affect the runtime performance.

Post-synthesis validation. In post-synthesis validation, Validator conducts static analysis over a synthesized binary. Unlike typical binary analysis that assumes a stripped binary, post-synthesis validation enables more sophisticated analyses by taking advantage of the metadata (e.g., the control-flow information) provided by Synthesizer. Post-synthesis validation takes in the form of validation passes coupled with each instrumentation pass. Based on the control-flow information, Validator executes validation passes at the basic-block level. Validator iterates through all functions in the binary and invokes a procedure implemented in each validation pass at the beginning of each basic block, which performs a series of checks based on the content of the basic block (i.e., raw bytes). For example, a procedure can determine whether specific instrumentation is applied based on pattern matching. If any of the validation passes fails, Validator rejects the binary. Optionally, the procedure can utilize other metadata such as the original IR instructions that map to the basic block to facilitate the analysis beyond binary scanning.

Final binary attestation. In addition to using remote attestation for hardware probing (§4.1), PRIDWEN uses remote attestation to attest the dynamically synthesized binary inside the enclave. SGX does not natively support the attestation of dynamic enclave content; instead, traditional remote attestation measures only the static code and data that are initially inside an enclave, which is used as a piece of evidence throughout the process of remote attestation. To attest dynamic content, PRIDWEN incorporates a two-step scheme that extends the attestation of static content.

Attack surface	SW-only Mitigation	HW-assisted Mitigation
Cache timing	Interrupt (Varys)	Cache flushing (microcode)
Page fault	Interrupt (Varys)	T-SGX
HT	Co-location (Varys)	HT disabling (microcode)
Speculative execution	QSpec	IBRS (microcode)
Static layout	ASLR	N/A

Table 3: Attack surfaces and software-only or hardware-assisted mitigation schemes PRIDWEN implements. CPUs with recent microcode update do not have some of the attack surfaces.

In the first step, a user uses the SGX standard procedure to attest the static part of PRIDWEN and establishes a secure channel [45]. Then, the user sends a Wasm binary `p.wasm` to PRIDWEN via the secure channel and PRIDWEN starts to synthesize the final binary based on the hardware configuration (`hw_config`) of the execution platform. In the second step, PRIDWEN sends the user: 1) the measurement of the synthesized binary `p.code` (i.e., the hash of native code blocks `hash(p.code)`) and 2) the `hw_config`. The user can then validate `hash(p.code)` thanks to a PRIDWEN’s property: *reproducible synthesis* (§4.3). With the reproducible synthesis, the user can validate the final binary based on both `p.wasm` and `hw_config`.

5 Implementation

We implement a prototype of PRIDWEN with 25k lines of C code on top of the Intel Linux SGX SDK 2.5.102. The size of PRIDWEN in binary is only 1.26 MiB, maintaining a slim footprint of trusted computing base (TCB). For native code generation, we implement an x86 backend to support the full Wasm instruction set.

Runtime support. Our prototype provides an Emscripten-compatible runtime support that allows to run fairly large, complex applications such as Lighttpd, as shown in §6. The application is directly compiled from unmodified C source code to a Wasm binary using the Emscripten compiler.

Attestation of synthesized binaries. Our prototype supports *final binary attestation* mentioned in §4.4. Also, the prototype provides a tool that allows users to locally validate the measurement of the synthesized binary.

5.1 Example Passes

To illustrate the capability and practicality of PRIDWEN, our prototype implementation integrates four SCA mitigation schemes (ASLR [43], Varys [26], T-SGX [24], and QSpec [30]) into PRIDWEN using provided APIs, and *simultaneously* cover five important SCA surfaces (cache timing, page fault, HT, speculative execution, and static layout) (Table 3) based on the probed hardware configuration (§4.1). Although the four mitigation schemes were not originally introduced by this work, they are selected to demonstrate how to integrate existing mitigation techniques using PRIDWEN. Because the control-flow information (including the definition of basic

blocks in the binary) is required by most of the passes, we implement a control-flow analysis pass to share the information with other passes, eliminating the overhead posed by repetitive analyses. PRIDWEN helps to prioritize hardware-assisted mitigations with lower overheads if the execution platform supports them (e.g., TSX for T-SGX), and safely avoid redundant countermeasures if the platform is free from the corresponding SCAs thanks to recent microcode or hardware updates [21, 22, 29, 69].

5.1.1 Example Pass #1: Fine-grained ASLR

Many SCAs rely on accurate memory layout information to improve the granularity of leaked information. Thus, PRIDWEN enables fine-grained ASLR by default, which randomizes the location of every basic block, as a general mitigation scheme against SCAs.

Integration. We adopt the similar compiler-level scheme from SGX-Shield [43] by inserting a `jmp` instruction at the end of every basic block. First, the pass uses the `onControlStart` and `onControlEnd` APIs (Table 2) to identify the structure of a basic block. Next, the pass inserts a `jmp` with a symbol that points to the succeeding basic block if it does not end with a `jmp`. The pass also updates the targets of other branches accordingly by using the `onMachineInstrEnd` API. Later, Synthesizer shuffles the placement of each basic block if the ASLR pass is enabled. During the relocation, the generated symbols allow Synthesizer to resolve the target of each branch to a basic block at a randomized location.

Post-synthesis validation. We integrate a validation pass that performs the following checks: 1) whether a basic block terminates with a `jmp` and 2) whether each branch points to the correct target based on the control-flow information.

5.1.2 Example Pass #2: T-SGX

SGX allows the OS to handle page faults. Page-fault SCAs [14] exploit this design decision by intentionally making enclave pages inaccessible and observing which pages are accessed. To defeat this SCA, T-SGX [24] hides page faults from the OS by running an enclave spitted as small code blocks inside TSX transactions. As a result, all page faults occurring during the execution are suppressed (i.e., not delivered to the OS).

Integration. Our T-SGX pass has cache usage and execution time analyzers for native instructions using the `onMachineInstrEnd` API. Based on the analysis results, the pass determines the scale of a code block. Next, the pass replaces branch instructions at the end of the block with the instructions redirecting to the springboard (i.e., a `lea` for saving the address of the next code block and a `jmp` to the springboard). Similar to the fine-grained ASLR pass, the T-SGX pass identifies basic blocks in the binary by using the `onControlStart` and `onControlEnd` APIs. To support the springboard, the pass places the springboard code before

the entry function (e.g., main) of the binary by using the `onFunctionStart` API.

Post-synthesis validation. The validation pass for T-SGX checks 1) the presence of the springboard, 2) the presence of the instructions to jump to the springboard at the end of every code block, and 3) whether the target of the instructions in step 2 correctly points to the springboard. Optionally, the pass can re-analyze cache usage and execution time to ensure the correctness of the code splitting.

5.1.3 Example Pass #3: Varys

SCAs usually require frequent interrupts or HT to accurately identify the execution context (e.g., which pages are accessed) or to attack in-core cache and speculation units. Varys [26] is a software-based approach to detect such behaviors.

High-frequency AEX detection and cache eviction. Varys identifies the occurrence and frequency of interrupts during the enclave execution by analyzing AEXs. Specifically, whenever an AEX occurs, SGX updates the corresponding field in the SSA. Thus, by counting the number of instructions executed at every basic block and periodically polling the SSA, Varys can estimate the frequency of AEXs. In addition, Varys explicitly evicts cache lines upon detecting AEXs to mitigate cache-based SCAs.

Co-location test. To prevent scheduling victim and attack threads to the same HT core, Varys prepares a pair of SGX threads and checks whether they are in the same core via an L1-cache-based covert timing channel. Varys performs this co-location test whenever it observes an AEX.

Integration. PRIDWEN's Varys pass inserts the checking code at the beginning of every basic block by using the `onControlStart` and `onControlEnd` APIs. Unlike the original Varys that counts the number the instructions at the LLVM IR level, our pass counts the number of native instructions with the help of the `onMachineInstrEnd` API. For the SSA polling routine, the pass inserts the code before the entry function of the binary via the `onFunctionStart` API. The pass also adds the co-location test code to the SSA polling routine (i.e., after detecting an AEX).

Post-synthesis validation. The validation pass verifies 1) the presence of the checking code, 2) the correctness of the instruction number added to the counter, 3) the presence of the SSA polling code, and 4) whether the target of the `call` in the checking code points to the SSA polling routine.

5.1.4 Example Pass #4: QSpectre

One software-based approach to mitigate the Spectre attack is to use serializing instructions (e.g., `lfence`) to prevent the CPU from speculatively executing instructions beyond the intended placements. Following this idea, Microsoft Visual Studio has adopted a compiler-based scheme, QSpectre [30],

```
1 [ { "name": "tsgx",
2   "sca": [ "page" ],
3   "dependency": { "hw": [ "tsx" ], "weak": [ "aslr" ] },
4   "priority": "high"
5 },
6 { "name": "cotest-tsgx",
7   "sca": [ "ht" ],
8   "dependency": { "hw": [ "ht" ], "strong": [ "tsgx" ] },
9   "priority": "high"
10 },
11 { "name": "qspectre",
12   "sca": [ "spectre" ],
13   "dependency": { "hw": [ "!ibrs", "ht" ] },
14   "priority": "high"
15 },
16 { "name": "varys",
17   "sca": [ "cache", "page" ],
18   "dependency": { "hw": [ "!cache" ] },
19   "priority": "medium"
20 },
21 { "name": "cotest-varys",
22   "sca": [ "ht" ],
23   "dependency": { "hw": [ "ht" ], "strong": [ "varys" ] },
24   "priority": "medium"
25 },
26 { "name": "aslr",
27   "priority": "high" } ]
```

Figure 3: Example pass selection policy P (a .json file). name: name of the current pass; sca: the SCAs to mitigate; dependency: the required hardware (hw) and dependent passes (can be weak or strong); priority: the priority of the current pass.

which finds potentially vulnerable code patterns and inserts `lfence` instructions During compilation.

Integration. Instead of inserting `lfence` instructions based on pattern matching, which can be bypassed [70], PRIDWEN's instrumentation pass for QSpectre adopts a simple, yet effective strategy: inserting `lfence` instructions to all `if-else` structures. More concretely, the pass inserts an `lfence` instruction right after the conditional branch in the code of an `if-else` structure. This pass uses the `onMachineInstrEnd` API and determines if a conditional branch is in an `if-else` structure by consulting the `CompilerContext` data structure.

Post-synthesis validation. The validation pass simply checks the presence of the `lfence` instruction in every `if-else` structure.

5.2 Pass Coordination

PassManager selects the optimal set of passes and resolves potential conflicts following the steps mentioned in §4.2. To incorporate the four passes into PRIDWEN, we specify the pass selection policy P (shown in Figure 3). Note that the policy P is just an example; PRIDWEN can transparently support any developer-provided policies by design.

Pass selection. First, PassManager selects all feasible passes according to hardware dependencies. For example, it selects the QSpectre pass if IBRS is disabled and HT is enabled (Line 13). Next, it prunes passes that target overlapping SCAs based on the priority. We prioritize T-SGX (hardware-based) over Varys (software-based) in P such that if PassManager has selected the T-SGX pass because TSX is available, it will omit the Varys pass. Then, PassManager checks the unprocessed

passes in P and includes those whose dependencies are all satisfied (e.g., co-location test for T-SGX `cotest-tsgx`), or without any dependency (e.g., ASLR).

Pass ordering. Based on the dependencies specified in P , PassManager also determines the order of applying passes to resolve potential conflicts among them. For example, the ASLR and T-SGX passes can compete with each other to instrument branches at the end of basic blocks. To avoid such conflicts, a weak dependency between the two passes is indicated in P (Line 3). Accordingly, PRIDWEN serves the ASLR pass first and then applies the T-SGX pass with slight modification (e.g., instrument `jumps` inserted by the ASLR pass to make it point to the T-SGX springboard). Also, PRIDWEN must apply the T-SGX co-location test pass `cotest-tsgx` after the T-SGX pass itself to correctly insert the testing code at the springboard, which is indicated in P as a strong dependency (Line 8). As passes without dependencies (e.g., QSpectre) can be applied at anytime, PRIDWEN simply applies them after serving passes with dependencies.

6 Evaluation

We evaluate PRIDWEN on successful mitigation of individual targeted SCAs (§6.1), the semantic correctness of the input Wasm program (§6.2), the performance characteristics of the PRIDWEN loader (§6.3), and the performance overhead of PRIDWEN-synthesized binaries (§6.4).

Experiment setup. We ran all the experiments on a machine with a 4-core Intel i7-6700K CPU (Skylake microarchitecture) operating at 4 GHz with 32 KiB L1 and 256 KiB L2 private caches, an 8 MiB L3 shared cache, and 64 GiB of RAM. The machine was running Linux kernel 4.15. The PRIDWEN loader is compiled with `gcc 5.4.0` and executed on top of the Intel Linux SGX SDK 2.5.102.

Applications and test suites. We use three real-world applications or libraries (Lighttpd 1.4.48 [71], libjpeg 9a [72], and SQLite 3.21.0 [73]) as a macro-benchmark suite representing large, complex applications, as well as a micro-benchmark suite, PolyBenchC [74]. The benchmark suite consists of 23 small C programs with only numerical computations (i.e., no `syscall`) that are used to evaluate the runtime performance of just-in-time compiled Wasm binaries against native C binaries [34]. We compile the original source code of each micro- or macro-benchmark program into Wasm using Emcripten [42], an LLVM-based compiler. We also directly port all of the programs using SGX SDK to serve as baseline versions. We use the official Wasm specification test suite [44] to test the correctness of the synthesis of PRIDWEN (§6.2).

Methodology. For each run of experiments, we take the compiled Wasm binaries as input to PRIDWEN. To evaluate PRIDWEN-synthesized binaries with distinct sets of defense schemes enforced, we manually configure PRIDWEN before each run. We use **BASE** to represent the configuration of base-

line compilation (i.e., synthesis without instrumentation) and the name of defense schemes to represent the configuration of enforcing the corresponding schemes. For example, **TSGX** indicates the configuration with T-SGX enforced. For the ease of comparing Varys and T-SGX, the rest of the section uses **VARYS**, to represent its original design with the co-location test, respectively. **QSpectre** or **QS** represent QSpectre. To measure the execution time of each application, we use the `rdtsc` instruction via an `0Ca11` inside an enclave. The reported results are averaged over 10 runs.

6.1 Security Analysis

In addition to statically checking the enforcement of each mitigation scheme via validation passes, we also manually verify whether the integrated versions of example passes are effective and compatible by running simplified SCAs against them and building a test suite (§6.2). After hardening a test binary over the SCA surfaces with different combinations, we introduce frequent page faults and interrupts, manipulate processor affinity, and run a simple Spectre attack [19]. We confirm that the T-SGX pass suppresses page faults during runtime, the Varys pass detects frequent interrupts and thread co-location (if HT is enabled), and the QSpectre pass disrupts speculation (if IBRS is disabled). In addition, combining the ASLR pass and frequent interrupt detection (the Varys pass) can effectively mitigate or slow down an attacker's attempts to infer the fine-grained memory layout.

6.2 Correctness

To validate whether the synthesized program behaves as expected, we use the official Wasm specification test suite [44], which provides comprehensive test cases for all Wasm instructions. The test suite consists of 73 programs. Each program includes a set of functions and test cases that specify the expected outputs with given inputs. We ran the test suite on PRIDWEN with all hardening configurations and reported the results in terms of *pass* or *fail* on each program. In addition to the test suite, we also record intermediate values of all benchmark programs (by manually inserting `printf`) for both baseline and PRIDWEN-synthesized version and compare them.

Results. The results show that programs with all hardening configurations successfully pass all the test cases, which indicates that 1) the baseline compilation of PRIDWEN (**BASE**) faithfully follows the specification of Wasm, and 2) the enforcement of schemes does not modify the behavior of the program. Moreover, there is no difference when comparing intermediate values between **BASE** and the synthesized binaries.

6.3 Performance of PRIDWEN

To show both runtime and memory overheads of the PRIDWEN loader, we measured the execution time that PRIDWEN takes to generate native C binaries and the additional memory that it allocates during the entire process (by

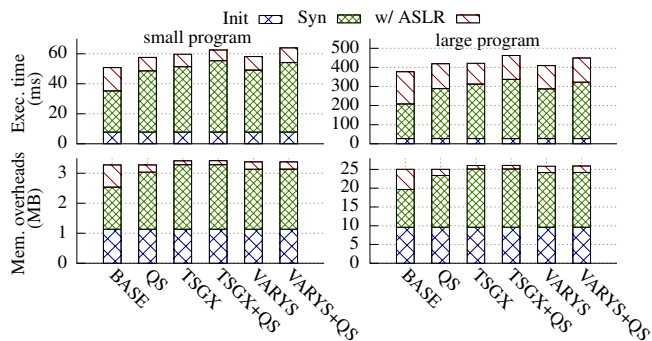


Figure 4: The top and bottom figures show runtime overheads and memory overheads, respectively, of PRIDWEN on program synthesis.

hooking `malloc`). To demonstrate the impact on the size of input, we used one small (2mm, 52 kB) and one large (`lighttpd`, 462 kB) Wasm binaries as inputs. We also ran experiments with different configurations of PRIDWEN to show the impact of enforcing different mitigations. As the co-location test depends on either T-SGX or Varys and requires only adding a piece of code to each scheme, we do not include it in the selected configurations. Note that the overhead only needs to be paid once during the first initialization and synthesis.

The results are shown in [Figure 4](#). We divide each bar into three parts: the initialization stage (blue), the synthesis stage (green), and additional overhead when the ASLR is enforced on top of the corresponding configuration (red). The initialization stage includes the time spent on hardware probing, PassManager initialization, and Wasm parsing. The synthesis stage represents the time spent on compilation and instrumentation in Synthesizer.

Runtime performance. For the runtime overhead ([Figure 4](#)), it is clear that given the same program, the execution time of the initialization stage is fixed regardless of the configurations. For the large program, PRIDWEN spends more time during the initialization stage, which is mostly due to the process of parsing the Wasm binary; however, the proportion of the execution time spent in the initialization stage decreases, which indicates that PRIDWEN spends more time on the synthesis stage for the large program. Also, enabling ASLR for the large program incurs higher overhead since it has more basic blocks. In addition, enforcing more schemes incurs higher overheads as expected. Overall, the one-time overhead of PRIDWEN is acceptable (less than 500 ms for the large program).

Memory overhead. For the memory overhead ([Figure 4](#)), the results show that PRIDWEN requires a fixed amount of memory during the initialization stage for the same program. PRIDWEN requires more memory for the large program, since the majority of the required memory is used to store the IR of the input program during the parsing process. We also observe similar memory requirements for PRIDWEN with the **BASE** configuration in the synthesis stage, since it needs more memory to maintain the metadata during compilation for the large program. Also, enabling ASLR on top of **BASE** incurs

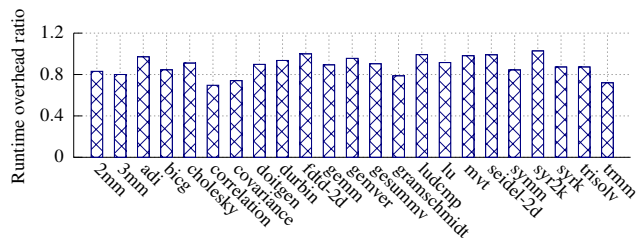


Figure 5: The runtime performance of PRIDWEN-synthesized Wasm programs compared to native C binaries.

the highest overhead. The reason is that the instrumentation passes of each scheme all depend on the pass that manages the control-flow graph (CFG) information. As the ASLR pass can share the CFG information with other passes and can directly reuse such information during runtime, enabling ASLR on top of them incurs less overhead; when enabling just ASLR in **BASE**, it needs to generate the CFG information itself, resulting in a large memory overhead. Note that the memory overhead is only imposed once before the execution of the synthesized binary; thus it does not affect the memory usage of the binary in run-time.

6.4 Performance of Synthesized Binaries

We measure the runtime and memory overheads of PRIDWEN-synthesized binaries. We compare the results with those of native C binaries ported directly into SGX enclaves. In addition, we measure the performance overhead of the PRIDWEN-synthesized version of the defense schemes by comparing the results with those of the **BASE** configuration, and match it to that of the original implementations (i.e., the overhead indicated in the original papers). We confirm that the overheads are mainly inherited from the original design of the countermeasures; PRIDWEN only imposes minimal amount of overheads in the binaries.

Runtime performance. [Figure 5](#) shows the results of running the PolybenchC with the **BASE** configuration, which are normalized to the execution time of the native C programs. Our results indicate that PRIDWEN-synthesized binaries have negligible slowdown or are even faster than the native C binaries, without any mitigation schemes enforced. The execution time of PRIDWEN-synthesized binaries are $0.7\times$ – $1.0\times$ of that of the native C binaries. Likewise, the very initial evaluation [34] of the in-browser Wasm compiler reports similar execution overhead results on PolybenchC programs ($0.5\times$ – $1.4\times$ of that of native C). The runtime performance of PRIDWEN-synthesized Wasm programs is comparable to or even better than that of C programs. This might be due to the small size of PolybenchC programs, with only numerical computations. Therefore, the synthesized Wasm programs are fairly compact and similar to native binaries. Furthermore, the difference between compilers (i.e., Emscripten and GCC) may also contribute to the results. When programs get more complex, performance overhead increases as their Wasm forms

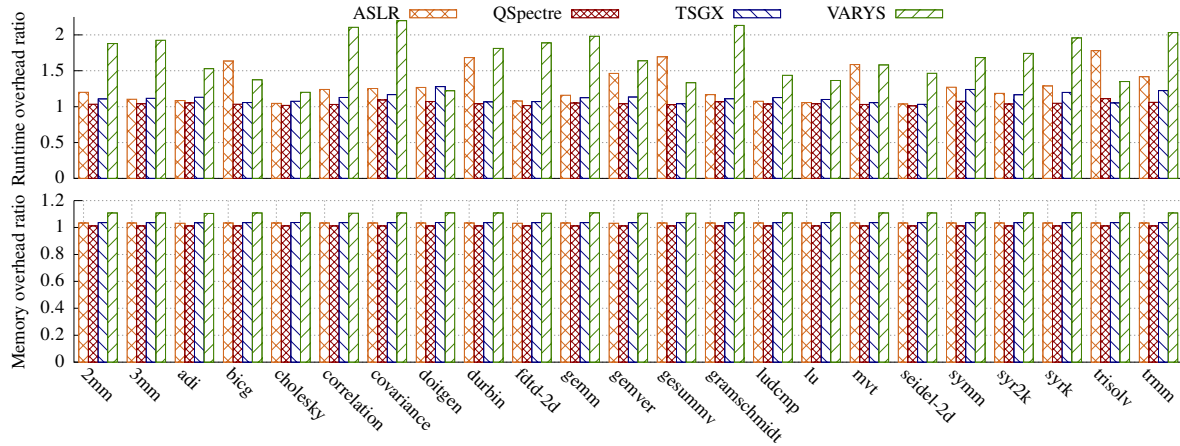


Figure 6: The top and bottom figures show the runtime performance and memory overheads, respectively, of PRIDWEN-synthesized programs secured with different mitigation schemes, compared to **BASE**.

no longer maintain the similarity to native binaries (e.g., the number of instructions with one-to-many mappings grows).

Figure 6 demonstrates the results of running PolybenchC with defense schemes enforced. The bar on the figure represents the relative execution time of the program to the **BASE** configuration. **ASLR** incurs various overheads due to different numbers of randomized basic blocks being executed, which is not cache-friendly. Similarly, **QSpectre** also incurs various but smaller overheads, which result from the number of `lfence` instructions being executed.

Regarding T-SGX and VARYS, **TSGX** incurs less overhead than **VARYS** does. Especially, **VARYS** suffers from high overhead when it needs to check AEXs inside a loop structure (see an example in [Appendix B](#)). **VARYS** cannot avoid this issue without compromising its security guarantees. In contrast, **TSGX** supports loop optimization, which puts an entire loop into single transaction when possible.

Memory overhead. **Figure 6** shows how much memory each mitigation demands on top of the binaries with **BASE**. On average, the memory overheads of the synthesized binaries are about $1.2\times$ compared to the baseline binaries, which is moderate.

Real-world applications. We use three real-world applications as case studies to show that PRIDWEN provides sufficient support for large, complex programs. In addition to the *Lighttpd* (a web server), the other two applications are based on *libjpeg* and *SQLite* libraries. The *libjpeg* application supports both compressing and decompressing a *jpeg* image and the *SQLite* application supports basic database operations, including insert, select, update, and delete. We use the HTTP benchmarking tool, *wrk*, for evaluating the throughputs of the *Lighttpd*. For the other two applications, we measure the execution time of each supported operation and report the average values over 10 runs.

Figure 7 shows the results of *Lighttpd*. The slowdown of **BASE** is $1.5\times$ to the native version. **TSGX** incurs less over-

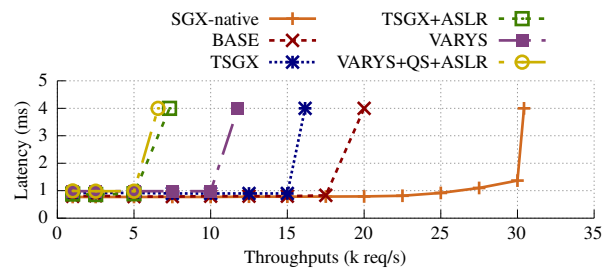


Figure 7: The performance of *Lighttpd*; QS: QSpectre.

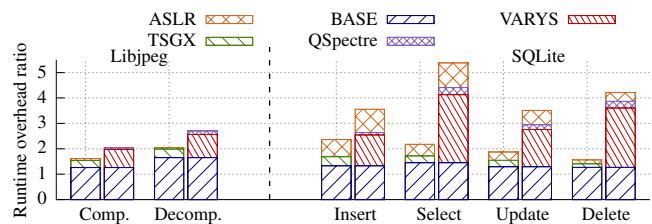


Figure 8: The performance of synthesized *libjpeg* and *SQLite*; each left bar illustrates hardware-assisted passes, while each right bar illustrates software-only passes.

heads compared to **VARYS** ($1.9\times$ versus $2.6\times$), demonstrating the advantage of hardware-assisted mitigation schemes over software-only ones. **ASLR** incurs significant overhead because *Lighttpd* has a large number of small-sized basic blocks; This shortens the gap between **TSGX** and **VARYS** when they are enforced with **ASLR**. When enforcing multiple mitigation schemes, the slowdown of *Lighttpd* is up to $4.6\times$ compared to the native binary.

Figure 8 presents the performance of *libjpeg* and *SQLite* applications. We use stacked bars to represent the incurred overheads when applying the optimal set of mitigations on top of a viable hardware configuration. The runtime overhead of **BASE** is $1.2\times$ – $1.7\times$ compared to the native versions. The overheads of individual mitigation schemes are similar to the results of PolyBenchC presented in **Figure 6** (e.g., **TSGX** in-

curs less overheads than **VARYS**). The average slowdown of hardware-assisted mitigation schemes is 1.9× while that of software-only mitigation schemes is 3.4×. Depending on hardware configurations, hardware-assisted mitigation schemes are 2.1×–5.4× faster than software-only ones.

Again, as PRIDWEN only aims to integrate multiple mitigation techniques, most of the performance overheads are inherited from the original design of the countermeasures, while PRIDWEN itself does not impose significant overheads.

7 Discussion

Adding new defenses to PRIDWEN. PRIDWEN is designed with future scenarios in mind. Thanks to the high-level instrumentation APIs provided by PRIDWEN (Table 2), new instrumentation-based defense techniques can be easily added to PRIDWEN as a pass. PRIDWEN pass APIs offer different instrumentation granularity with sufficient information about the corresponding instruction on both IR- and native-level, which should meet all instrumentation needs. We recommend to develop new defenses directly with PRIDWEN to save the hassle of replacing heterogeneous instrumentation APIs with PRIDWEN versions during integration. The developer will also need to provide the type of side-channel attack targeted by the new defense, and the possible dependency on specific hardware features or existing techniques in the pass selection policy. This allows Prober and PassManager to resolve potential incompatibility issues and conflicts, and correctly enforce the new defense. Compared to the effort needed to write a new pass, writing a pass selection policy should be much simpler.

Upgrade PRIDWEN. Modern hardware is evolving quickly with updated features useful for security purposes and PRIDWEN is designed to keep up with the hardware evolution. It is necessary for PRIDWEN to allow probing of new hardware features for defense techniques built with such features. The probing logic for the specific hardware feature will need to be added by PRIDWEN developers using either exception-based instruction probing or trusted remote attestation (§4.1) to bypass untrusted privileged software. Novel and secure techniques are also welcomed to probe the hardware capability of the platform. We hope that PRIDWEN can motivate hardware manufacturers to provide official secure probing helpers for hardware features. Upgrade of PRIDWEN is the effort of both the hardware and the software communities. The power of PRIDWEN is truly unleashed when equipped with the most updated inclusion of hardware features and mitigation schemes.

The recent SmashEx attack [75]. A recent paper presents the SmashEx attack targeting the SGX SDKs; the targeted SDKs do not properly handle *re-entrancy* in their asynchronous exception handling logic, which allows the attacker to compromise the integrity of the SSA region and modify GPRSGX.RIP. PRIDWEN is built on top of the trusted Intel

SGX SDK. The vulnerability of asynchronous exception handling in SGX (SmashEx) is rooted in the SGX SDK, and it was already mitigated by recent patches [76, 77]. That is, the enclave-specific SSA that hosts the GPRSGX.RIP cannot be compromised with the attack in the latest SDK. PRIDWEN should work with the latest SGX SDK because it does not heavily rely on or modify a certain SDK version.

Program synthesis on the cloud side vs. the user side.

PRIDWEN makes the design decision of conducting program synthesis on the cloud side to minimize the extra effort required for preparation on the user side. An alternative solution would be deploying a dedicated program on the cloud to report the hardware configuration back to the user, then the user in return synthesizes and caches the hardened binary themselves and sends the binary to the cloud for deployment. Synthesizing and caching the binaries of different configurations at the user side can save compilation cost on the cloud side; however, this puts more burden on the user. In addition, it would be difficult to update the binaries on the cloud side when the configuration changes, since the server has to wait for the user to compile and transmit the updated version. In contrast, when the configuration on the server is changed (e.g., after reboot), PRIDWEN automatically re-synthesizes the application and applies the change upon restarting. It is also possible to optimize PRIDWEN in a similar way by caching the synthesized programs of different configurations on the cloud side to reduce the compilation overhead.

8 Conclusion

PRIDWEN is a framework to dynamically synthesize a secure SGX program that is optimally hardened against various SCAs simultaneously, while preventing any *deployability*, *redundancy*, or *incompatibility* problem. To overcome the restrictions of the static deployment model of SGX, PRIDWEN adopts Wasm as the IR, and supports smooth integrations of instrumentation passes for both hardware-assisted and software-only mitigations. PRIDWEN selects an optimal set of mitigations to be applied at runtime according to the hardware configurations of the target platform, and provides means for the user to validate and attest the final synthesized binary. We implement a prototype of PRIDWEN, which integrates four SCA defenses. Through extensive evaluation, we show that PRIDWEN efficiently hardens SGX programs with chosen defenses, while incurring moderate performance overhead.

9 Acknowledgment

We would like to thank the anonymous reviewers and our shepherd for their helpful feedback. We also would like to thank Scott Constable and Yuan Xiao from Intel for constructive discussions. This research was funded by Intel and the NSF award NSF-1563848.

References

- [1] J. Mangalindan, “Is User Data Safe in the Cloud?” <http://tech.fortune.cnn.com/2010/09/24/is-user-data-safe-in-the-cloud>, September 2010.
- [2] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, “Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, Nov. 2009.
- [3] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, “Using Innovative Instructions to Create Trustworthy Software Solutions,” in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, Tel-Aviv, Israel, 2013, pp. 1–8.
- [4] Intel, “SGX Tutorial, ISCA 2015,” <http://sgxisca.weebly.com/>, Jun. 2015.
- [5] N. Porter, “Introducing Asylo: an open-source framework for confidential computing,” 2018, <https://cloud.google.com/blog/products/gcp/introducing-asylo-an-open-source-framework-for-confidential-computing>.
- [6] Microsoft, “Open Enclave SDK,” 2019, <https://openenclave.io/sdk/>.
- [7] Microsoft Azure, “Azure Confidential Computing,” 2019, <https://azure.microsoft.com/en-us/solutions/confidential-compute/>.
- [8] S. Johnson, “Intel SGX and Side-Channels,” <https://software.intel.com/en-us/articles/intel-sgx-and-side-channels>.
- [9] F. Brasser, U. Müller, A. Dmitrienko, K. Kostinen, S. Capkun, and A. Sadeghi, “Software Grand Exposure: SGX Cache Attacks Are Practical,” in *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, Vancouver, BC, Canada, Aug. 2017.
- [10] M. Hähnel, W. Cui, and M. Peinado, “High-Resolution Side Channels for Untrusted Operating Systems,” in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017.
- [11] F. Dall, G. D. Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom, “CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks,” in *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2018.
- [12] A. Moghimi, T. Eisenbarth, and B. Sunar, “MemJam: A false dependency attack against constant-time crypto implementations in SGX,” in *Cryptographers’ Track at the RSA Conference*. Springer, 2018.
- [13] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bind-schaedler, H. Tang, and C. A. Gunter, “Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX,” in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct.–Nov. 2016.
- [14] Y. Xu, W. Cui, and M. Peinado, “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems,” in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.
- [15] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, “Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution,” in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.
- [16] S. Weiser, R. Spreitzer, and L. Bodner, “Single Trace Attack Against RSA Key Generation in Intel SGX SSL,” in *Proceedings of the 13th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Seoul, South Korea, Jun. 2018.
- [17] J. Gyselinck, J. Van Bulck, F. Piessens, and R. Strackx, “Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution,” in *International Symposium on Engineering Secure Software and Systems*. Springer, 2018, pp. 44–60.
- [18] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, “Preventing Your Faults From Telling Your Secrets,” in *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Xi’an, China, May–Jun. 2016.
- [19] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, “SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution,” in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 142–157.
- [20] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, “Spectre returns! speculation attacks using the return stack buffer,” in *Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT)*, Baltimore, MD, Aug. 2018.
- [21] Intel, “Q3 2018 Speculative Execution Side Channel Update,” 2018, <https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html>.

- [22] —, “Intel Side Channel Vulnerability MDS,” 2019, <https://www.intel.com/content/www/us/en/architecture-and-technology/mds.html>.
- [23] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, “Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory,” in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.
- [24] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, “T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs,” in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.
- [25] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, “Racing in hyperspace: Closing hyper-threading side channels on SGX with contrived data races,” in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.
- [26] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, “Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks,” in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jul. 2018.
- [27] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, “Detecting privileged side-channel attacks in shielded execution with Déjà Vu,” in *Proceedings of the 12th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Abu Dhabi, UAE, Apr. 2017.
- [28] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, “Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing,” in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.
- [29] Intel, “Branch Target Injection / CVE-2017-5715 / INTEL-SA-00088,” 2018, <https://software.intel.com/security-software-guidance/software-guidance/branch-target-injection>.
- [30] A. Pardoe, “Spectre mitigations in MSVC,” 2018, <https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/>.
- [31] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, “Obfuscuro: A Commodity Obfuscation Engine on Intel SGX,” in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.
- [32] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostianen, and A.-R. Sadeghi, “DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization,” in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2019.
- [33] GCC team, “Using the GNU Compiler Collection (GCC): x86 Built-in Functions,” 2019, <https://gcc.gnu.org/onlinedocs/gcc/x86-Built-in-Functions.html>.
- [34] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, “Bringing the Web up to Speed with WebAssembly,” in *Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Barcelona, Spain, Jun. 2017.
- [35] WebAssembly Community Group, “WebAssembly Specification: Release 1.0,” Tech. Rep., May 2019.
- [36] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin, “Towards Memory Safe Enclave Programming with Rust-SGX,” in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.
- [37] W. Qiang, Z. Dong, and H. Jin, “Se-Lambda: Securing Privacy-Sensitive Serverless Applications Using SGX Enclave,” in *International Conference on Security and Privacy in Communication Systems (SecureComm)*, 2018.
- [38] Red Hat, “Enarx,” 2019, <https://enarx.io>.
- [39] Intel, “WebAssembly Micro Runtime,” 2019, <https://github.com/bytedcodealliance/wasm-micro-runtime>.
- [40] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, “SEISMIC: SEcure in-lined script monitors for interrupting cryptojacks,” in *European Symposium on Research in Computer Security*. Springer, 2018, pp. 122–142.
- [41] D. Lehmann and M. Pradel, “Wasabi: A Framework for Dynamically Analyzing WebAssembly,” in *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, Apr. 2019.
- [42] “emscripten,” 2015, <https://emscripten.org/>.
- [43] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, “SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs,” in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

- [44] “Mirror of the spec testsuite,” 2019, <https://github.com/WebAssembly/testsuite>.
- [45] Intel, “Code Sample: Intel Software Guard Extensions Remote Attestation End-to-End Example,” 2018, <https://software.intel.com/en-us/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example>.
- [46] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, “Lest we remember: cold-boot attacks on encryption keys,” *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.
- [47] Intel, “Exception Handling in Intel Software Guard Extensions (Intel SGX) Applications,” 2019.
- [48] —, “Intel 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4,” May 2019.
- [49] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution,” in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.
- [50] J. Van Bulck, F. Piessens, and R. Strackx, “Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic,” in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.
- [51] W. He, W. Zhang, S. Das, and Y. Liu, “Sgxlinger: A new side-channel attack vector based on interrupt latency against enclave execution,” in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 108–114.
- [52] D. Evtvushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, “BranchScope: A New Side-Channel Attack on Directional Branch Predictor,” in *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Williamsburg, VA, Mar. 2018.
- [53] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, “RIDL: Rogue in-flight data load,” in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2019.
- [54] M. Schwarz, M. Lipp, D. Moghimi, J. V. Bulck, J. Stecklina, T. Prescher, and D. Gruss, “ZombieLoad: Cross-Privilege-Boundary Data Sampling,” in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.
- [55] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. V. Bulck, and Y. Yarom, “Fallout: Leaking Data on Meltdown-resistant CPUs,” in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.
- [56] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin, “SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults,” in *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2017.
- [57] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, “Oblivious Multi-Party Machine Learning on Trusted Processors,” in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.
- [58] O. Ohrimenko, C. F. Manuel Costa, S. Nowozin, A. Mehta, F. Schuster, and K. Vaswani, “SGX-Enabled Oblivious Machine Learning,” in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.
- [59] S. Sasy, S. Gorbunov, and C. W. Fletcher, “ZeroTrace: Oblivious Memory Primitives from Intel SGX,” in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.
- [60] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, “Obliviate: A Data Oblivious File System for Intel SGX,” in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.
- [61] P. Zhang, C. Song, H. Yin, D. Zou, E. Shi, and H. Jin, “Klotski: Efficient Obfuscated Execution against Controlled-Channel Attacks,” in *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, Apr. 2020.
- [62] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, “Hacking in Darkness: Return-oriented Programming against Secure Enclaves,” in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.
- [63] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, “The Guard’s Dilemma: Efficient Code-Reuse Attacks against Intel SGX,” in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

- [64] Intel, “3rd Gen Intel Xeon Scalable processors,” <https://www.connection.com/~-/media/pdfs/brands/i/intel/intel-icelake-ds.pdf?la=en>.
- [65] —, “Attestation Service for Intel Software Guard Extensions (Intel SGX): API Documentation (Revision: 4.1),” 2018.
- [66] Greg, “SGX Attestation results in CONFIGURATION_NEEDED,” 2018, <https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/798777>.
- [67] —, “GROUP_OUT_OF_DATE - what is the most recent microcode version?” 2018, <https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/755769>.
- [68] Clemens Hammacher, “Liftoff: a new baseline compiler for webassembly in v8,” 2018, <https://v8.dev/blog/liftoff>.
- [69] A. Shilov, “Intel’s New Core and Xeon W-3175X Processors: Spectre and Meltdown Security Update,” 2018, <https://www.anandtech.com/show/13450/intels-new-core-and-xeon-w-processors-fixes-for-spectre-meltdown>.
- [70] P. Kocher, “Spectre Mitigations in Microsoft’s C/C++ Compiler,” 2018, <https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html>.
- [71] “Lighttpd,” 2003, <https://www.lighttpd.net/>.
- [72] “libjpeg,” 1991, <https://libjpeg.sourceforge.net/>.
- [73] “SQLite,” 2000, <https://www.sqlite.org/index.html>.
- [74] “PolyBench,” 2015, <http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/>.
- [75] J. Cui, J. Z. Yu, S. Shinde, P. Saxena, and Z. Cai, “Smashex: Smashing sgx enclaves using exceptions,” in *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, Virtual Event, Republic of Korea, Nov. 2021.
- [76] Intel, “The latest security information on Intel products,” 2021, <https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00548.html>.
- [77] Open Enclave, “Open Enclave SDK Elevation of Privilege Vulnerability,” 2021, <https://github.com/openenclave/openenclave/security/advisor/GHSA-mj87-466f-jq42>.
- [78] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, “VC3: Trustworthy data analytics in the cloud using SGX,” in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.
- [79] E. Bauman, H. Wang, M. Zhang, and Z. Lin, “SGXElide: Enabling Enclave Code Secrecy via Self-modification,” in *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO)*, Vienna, Austria, Feb. 2018.
- [80] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, “Ryoan: A distributed sandbox for untrusted computation on secret data,” in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.
- [81] A. Baumann, M. Peinado, and G. Hunt, “Shielding applications from an untrusted cloud with Haven,” in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.
- [82] C.-C. Tsai, D. E. Porter, and M. Vij, “Graphene-SGX: A practical library OS for unmodified applications on SGX,” in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017.
- [83] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. R. Pietzuch, “SGX-LKL: Securing the Host OS Interface for Trusted Execution,” *CoRR*, vol. abs/1908.11143, 2020. [Online]. Available: <http://arxiv.org/abs/1908.11143>
- [84] S. Arnautox, B. Tarch, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O’Keefe, M. L. Stillwell, D. Goltzsche, D. Eysers, R. Kapitza, P. Pietzuch, and C. Fetzer, “SCONE: Secure Linux containers with Intel SGX,” in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.
- [85] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, “Panoply: Low-TCB Linux applications with SGX enclaves,” in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

A Additional Related Work

In-enclave loader. Researchers study in-enclave loaders to enhance the security and deployability of Intel SGX. For the security, several loaders leverage randomization and encryption. SGX-Shield [43] loads SGX applications while

<pre> 1 # Varys 2 BB: 3 ... 4 jmp loop.header 5 loop.body: 6 call varys_check 7 ... 8 incq %rcx 9 loop.header: 10 call varys_check 11 ... 12 cmpq \$100, %rcx 13 jbe loop.body 14 loop.end: 15 call varys_check 16 ... </pre>	<pre> 1 # T-SGX 2 BB: 3 ... 4 leaq loop.header(%rip), %r15 5 jmp springboard.next 6 loop.body: 7 ... 8 incq %rcx 9 loop.header: 10 ... 11 cmpq \$100, %rcx 12 jbe loop.body 13 leaq loop.end(%rip), %r15 14 jmp springboard.next 15 loop.end: 16 ... </pre>
---	---

Figure 9: The comparison of Varys and T-SGX on a loop structure.

enforcing fine-grained ASLR. VC3 [78] and SGXElide [79] deploy encrypted SGX code while decrypting it within an enclave. Obfuscuro [31] obfuscates SGX code with Oblivious RAM. For the deployability, some loaders abstract the interface between SGX code and the outside. Ryoan [80] implements a two-way sandbox to securely execute untrusted code inside an enclave. Haven [81], Graphene-SGX [82], and SGX-LKL [83] run a library OS inside an enclave to execute unmodified programs. Similarly, SCONE [84] abstracts system call interfaces and Panoply [85] abstracts POSIX interfaces to run unmodified programs with SGX. Unlike these approaches, PRIDWEN focuses on how to instrument SGX applications according to the hardware features to improve their security.

SGX and Wasm. To the best of our knowledge, there are a few initial efforts to execute Wasm interpreters inside an enclave. Rust-SGX [36] can be configured to use Wasm as a backend. Se-Lambda [37] executes serverless functions written in Wasm inside an enclave. Also, Intel and Red Hat are developing Wasm runtime for SGX [38, 39]. However, unlike PRIDWEN, these approaches only run existing Wasm interpreters without improving their functionalities.

Wasm instrumentation. Other studies also instrument Wasm binaries to detect security attacks. SEISMIC [40] instruments Wasm binaries to inject an inline monitor for detecting cryptojacking. Wasabi [41] is a Dynamic Binary Instrumentation (DBI) tool that statically instruments Wasm binaries to inject hooks and dynamically runs JavaScript-based analysis code on them to find potential bugs. However, unlike PRIDWEN, they do not consider instrumenting native binaries compiled from Wasm binaries, which is necessary to adopt low-level security mitigations sensitive to machine code.

B Loop Comparison: Varys vs. T-SGX

The comparison of Varys and T-SGX on a loop structure is shown in [Figure 9](#).

TETRIS: Memory-efficient Serverless Inference through Tensor Sharing

Jie Li

College of Intelligence & Computing (CIC), Tianjin University
Tianjin Key Lab of Advanced Networking (TANKLAB)

Laiping Zhao*

CIC, Tianjin University
TANKLAB

Yanan Yang

CIC, Tianjin University, TANKLAB

Kunlin Zhan

58.com

Keqiu Li

CIC, Tianjin University, TANKLAB

Abstract

Executing complex, memory-intensive deep learning inference services poses a major challenge for serverless computing frameworks, which would densely deploy and maintain inference models at high throughput. We observe the excessive memory consumption problem in serverless inference systems, due to the large-sized models and high data redundancy.

We present TETRIS, a serverless platform catered to inference services with an order of magnitude lower memory footprint. TETRIS’s design carefully considers the extensive memory sharing of runtime and tensors. It supports minimizing the runtime redundancy through a combined optimization of batching and concurrent execution and eliminates tensor redundancy across instances from either the same or different functions using a lightweight and safe tensor mapping mechanism. Our comprehensive evaluation demonstrates that TETRIS saves up to 93% memory footprint for inference services, and increases the function density by 30× without impairing the latency.

1 Introduction

Serverless computing has seen its popularity explode in recent years, due to the ease of use, cost efficiency, resource management-free, and autoscaling advantages. Serverless inference, i.e., deploying deep learning (DL) inference services atop a serverless platform, is continuously adopted in the backend of the internet of things (IoT), mobile and web applications [1, 6, 11, 65, 74]. Some typical explorations include the Amazon Alexa [5], Facebook Messenger bot [49], and Netflix media transformation [38].

Inference services are commonly memory-intensive, consuming a substantial amount of memory throughout the computation. If performing inference in a serverless environment, we find that the current serverless platforms (e.g., AWS Lambda [2]) do not support inference computation well. First,

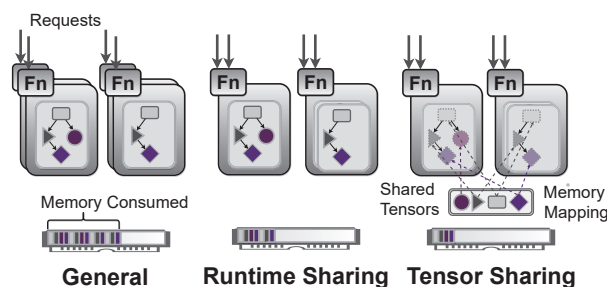


Figure 1: Schematic overview of memory consumption in serverless inference systems.

they cannot accommodate large inference models. For example, AWS Lambda limits the memory footprint of functions to $\leq 10\text{GB}$ [2], whereas the recent MT-NLG language model [36] even needs 2TB memory to load its 530 billion parameters. Second, these platforms cause a significant waste of memory resources. The "one-to-one mapping policy" of request to function instance by commercial platforms [2] introduces significant memory redundancy. The memory footprint of language runtime, libraries, and even tensors is repeated across function instances. On the other hand, serverless functions are usually *short-lived*. While the CPU processing of inference requests takes only a short time (e.g., 75% execute for less than 10 seconds in Microsoft Azure [60]), the memory is kept occupied without working for long due to the early reservation or keep-alive caching after completion (e.g., 15-60 minutes in AWS Lambda) [20, 58, 60]. Hence, how to improve the memory efficiency of serverless inference has become a crucial problem.

To reduce the memory footprint, prior works have proposed the *runtime sharing* method [16] (Figure 1), i.e., multiple requests share the same function instance runtime with increased instance concurrency. In serverless inference, *runtime sharing* can also be enabled through *batching*, which aggregates multiple inputs into a single batch submission for more efficient execution [1, 71, 74]. We study the memory footprint of serverless inference and observe that the *tensor redundancy* problem (i.e., tensors in the computational graphs of infer-

*Corresponding author: laiping@tju.edu.cn.

ence models are highly duplicated across function instances) still severely degrades memory efficiency. Such duplication is commonly caused by either replicated function instances or identical parts generated by pervasive pre-trained models or transfer learning [15, 52, 61, 62, 73]. We summarize 768 machine learning (ML) models utilized by 58.com, the China’s largest local life service website, finding that tensor redundancy exists in 67.5% of natural language processing models, 26.7% of image classification models, 30.1% of recommendation models.

The *tensor redundancy* problem can be mitigated through operating system kernel-level page merging methods (e.g., [3, 33, 47]), which scan for duplicate content in the memory and merge the content into a single physical page. However, they incur nonnegligible scanning overhead (e.g., 5 minutes scanning time) and thereby work well only for deduplicating fairly static memory pages, contradicting with *short-lived* nature of serverless functions. Since the scanning process occurs after applications are loaded, they can neither accelerate the function startup nor solve the *startup failure* problem caused by the out-of-memory (OOM) error. Moreover, their implementations require modifications to the operating system kernel, which is heavy and may introduce risks of side-channel attacks [42].

In this paper, we explore improving the memory efficiency of serverless inference system through both *runtime-level sharing* and *tensor-level sharing*. As illustrated in Figure 1, *the colocated function instances share tensor-memory pages with identical content*. Tensor sharing introduces several challenges that need to be addressed. (1) *Non-harming performance*: The sharing method should not impair the inference latency. (2) *Safe sharing*: The sharing process should not introduce data leakage risks. (3) *First-time sharing*: The sharing method should start working as long as the inference functions are activated, to accelerate the function startup and reduce the OOM errors. (4) *Low overhead*: The sharing method should be lightweight for easy integration with serverless frameworks, and transparent to tenants.

To overcome these challenges, we build TETRIS, a domain-specific serverless platform that caters to DL inference as backend-as-a-service (BaaS) offerings with high memory efficiency. For example, *after using TETRIS, the memory footprint of the LaBSE model (a multilingual BERT embedding model [18]) reduces from 1.97GB to merely 141.7MB, and the instance density also increases from 64 to 911 per 128GB-server without any modifications to the model*. TETRIS provides a complete solution for the memory bottleneck problem in serverless inference through automating runtime sharing, tensor sharing, memory reclaiming, and instance scheduling. It enables tensor sharing through a user-space page deduplication method, which has no pollution to the underlying systems and is thus incredibly lightweight and easy to implement. It is also highly efficient and supports to guarantee the Service Level Objectives (SLOs). DL developers would significantly

benefit from TETRIS, as it can save tremendous memory as well as monetary costs, or the freed memory can be reused for other purposes like caching.

Our contributions can be summarized as follows:

- We observe the *tensor redundancy* problem in serverless inference systems and propose the corresponding *tensor sharing* idea for memory efficiency.
- We design a lightweight, user-space tensor mapping-based sharing method, eliminating the tensor redundancy problem in serverless inference systems.
- We implement a prototype system of TETRIS, which is built with the open-source OpenFaaS [17] and TensorFlow Serving [51], supporting memory sharing, memory reclaiming, and instance scheduling.
- We extensively evaluate TETRIS using a comprehensive set of benchmarks and production workloads. The experimental results reveal that TETRIS can save up to 93% of memory and increase the function density by 30×, compared to the state-of-the-art approaches.

The rest of the paper is structured as follows. We study the data redundancy problem in serverless inference (§2), and use the findings to guide the design (§3) and implementation (§4) of TETRIS. We then evaluate the performance of TETRIS (§5), discuss related work (§6), and conclude (§7).

2 Background and Motivation

2.1 Inference on Serverless

The use of machine learning is divided into two phases: training and inference. While model parameters are continuously updated throughout the training phase, they remain fixed during the inference phase. In DL frameworks (e.g., TensorFlow), models are organized as computational graphs, and data in the graph are stored as *tensors*. The nodes represent operators or variables and edges denote the direction in which the tensor flows. In particular, the variable nodes of parameters and intermediate tensors produced on edges consume the most memory. The parameterized tensors may also be identical across different models, due to the pervasive usage of pretraining and transfer learning. As an example, we investigate the VGGish-based audio classification models that assist the telephone customer service in the local life service website, which are used for detecting various noises, non-human voices, dialects, etc. We find that these models share the bottom embedding layers, accounting for over 90% of the model parameters.

In serverless inference, each deep learning model is typically deployed into a separate function instance (e.g., a Docker container [45]). The function instances automatically scale in or out according to the fluctuation of requests. In the case of scaling out, an identical instance is created and its runtime, library, and model parameters are loaded into the memory repeatedly.

2.2 Motivations

To improve memory efficiency, we study the memory footprint of typical serverless inference models. The benchmarks are selected from TensorFlow Hub [22], with an average of 6.8k downloads. All of them are implemented and deployed on an OpenFaaS [17] testbed (Table 3).

Observation #1: Memory-intensive startup: *The loading of massive model parameters dominates the inference function instance startup.*

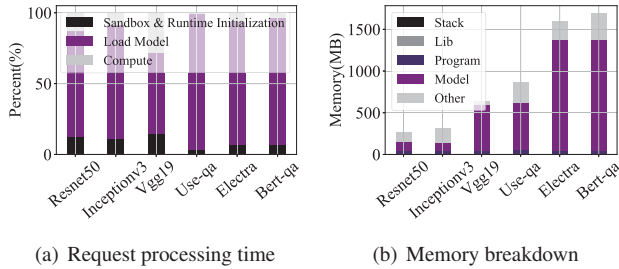


Figure 2: Request processing time and memory breakdown for various inference models.

The DL inference functions require numerous model parameters to load at startup, typically saved in a serialized model file. In particular, before the function can conduct the inference, the model loading thread continuously reads the parameter tensors from the disk and then deserializes and populates them into the corresponding node in the computational graph. With the memory page cache enabled to accelerate the startup, we measure the time of the three phases during request processing (Figure 2(a)): *sandbox and runtime initialization*, *model loading*, and *inference computation*. The *function startup* time is significantly longer than the *inference computation* time. Especially in the Bert-QA model case (with 1,300MB parameters), only 3.73% of the time is spent on computation. Even for the small model of Resnet50 (with 102MB parameters), the inference computation time still only accounts for 13.37%. If merely considering the startup, the majority of time is further spent on loading model parameters. For example, the ratios exceed 89% for both Electra and Bert-QA.

Observation #2: Memory-intensive computing: *The inference computation requires a substantial amount of memory, with the model parameters consuming the most.*

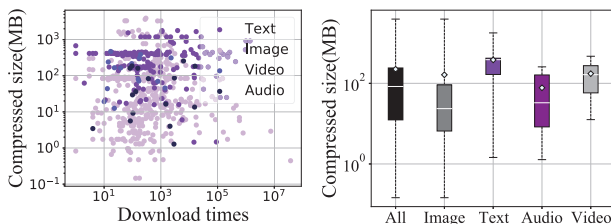


Figure 3: Analysis of the compressed model size and download times of deep learning models from TensorFlow Hub.

The memory footprint of an inference process can be divided into the following sections: *program code*, *libraries*, *the model* (for loading parameters in the form of tensors), *the function call stack*, *intermediate tensors* (i.e., generated at runtime) and *the network buffers* (i.e., allocated for receiving function requests). The *model* itself consumes the majority of memory.

We measure the memory consumption of different types of inference models at runtime and present the results in Figure 2(b). The model parameters occupy a major portion of all memory consumption (e.g., for VGG19, storing the parameter tensors accounts for more than 93% of the total memory consumption). We also compile a list of 625 machine learning models from TensorFlow Hub [22] and analyze their compressed model file sizes (Figure 3). In addition, 42% of the compressed model file sizes surpass 100MB, such as the popular text-embedding model *universal sentence encoder* (downloaded 1.4M times), which has a compressed model size of 0.89GB and consumes 1.72GB memory at runtime. Typically, the text processing models employing large embedding layers are substantially larger than the image, audio, and video processing models.

Observation #3: Runtime redundancy: *The inference runtime is replicated in memory due to the multilaunched instances. While both concurrent execution and batching enable runtime sharing, how to use them in combination is challenging.*

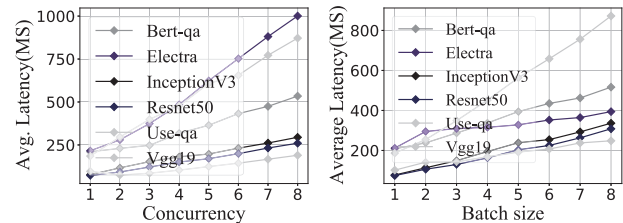


Figure 4: Average inference latency increases over batch size and concurrency.

Multilaunched instances lead to duplicated runtime memory consumption. The memory footprint can be reduced by packing multiple requests into an instance for sharing the runtime resources (e.g., ML model, framework, and network buffers.). In serverless inference, runtime sharing can be implemented in two ways: *batching* and *concurrent execution*. Given the same benchmarks and resource configurations, we evaluate the inference latency under both implementations. Figure 4 illustrates that both implementations increase the latency significantly: *batching* increases the latency due to the increased computing load, whereas *concurrent execution* increases latency due to the increased resource contention among threads, i.e., as concurrent requests compete for the computing threads in TensorFlow, the interleaved computation of operators increases the average latency.

We further measure memory consumption under various combinations of *batching* and *concurrency* (Figure 5). In par-

ticular, we generate a constant 200 requests per second (RPS) toward DenseNet169 [30] and Lstm [27] and specify their latency SLOs at 200ms and 30ms, respectively. After omitting combinations that do not guarantee the SLOs, we find that solely increasing either the concurrency or batch size leads to sub-optimal memory efficiency. For DenseNet169, increasing only the batch size quickly causes SLO violations due to the long waiting time in the batch queue, whereas increasing the concurrency does not. For the Lstm model, a larger batch size ($batchsize = 6$) achieves the least memory consumption under the SLO guarantee. As various combinations of batch size and concurrency may lead to completely different memory efficiency, selecting the best among them is essential.

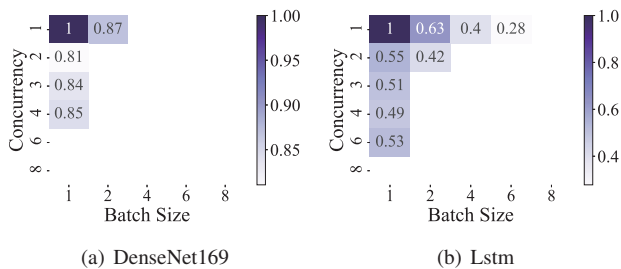


Figure 5: Normalized memory consumption of function instances under various combinations of batch size and concurrency.

Observation #4: Tensor redundancy: *The tensors of the constants, model parameters, are extensively replicated across function instances.*

Besides the runtime redundancy, parameterized tensors are also replicated across instances from the same function, i.e., *tensor redundancy*. We summarize 768 DL models at 58.com, finding that tensor redundancy also commonly exists across distinct functions. There are presently 27 business lines in 58.com, including jobs, housing, vehicles, cellphones, home services, resumes, etc. These businesses have common demands on DL inferences (e.g., image, text, video, and audio processing) but with different datasets (e.g., pictures of houses, cars, people, and smartphones). Primarily, there are two scenarios that cause tensor redundancy across distinct functions [15, 40, 52, 61, 62, 73]:

(1) *Multi-versioned functions.* In production scenarios, a DL model is frequently reused directly in various business contexts. As online web services have strict latency requirements (e.g., < 100 milliseconds), models with pipeline dependencies are commonly deployed together within a function to avoid the excessive network communications. In such cases, although these pipelines vary across businesses, tensors within the shared models stay identical. For example, the Optical Character Recognition (OCR) pipeline and the image moderation pipeline share the same text line identification and recognition models (e.g., Resnet), whereas the image moderation pipeline incorporates an extra model for keyword detection. Moreover, DL models are also commonly deployed

with specific pre- and post-processing modules for processing data in different formats, thus generating multi-versioned model pipelines.

(2) *Pretraining & transfer learning.* It refers to training a model with massive datasets for one task, where the learned parameters could be reused in other related tasks. In such way, new models could significantly benefit from prior knowledge to accomplish new tasks rather than from scratch. At 58.com, pre-trained models or transfer learning are widely used for reducing development costs. For example, in the house leasing business, webchats are recorded and different DL models are called to identify the status of the landlord (e.g., whether the house was leased) and tenant (e.g., a genuine renter or housing agent) respectively. In the telephone customer services, there are also distinct models to assess the recruitment, rental and housekeeping intentions of users, respectively. These models with similar tasks are all built from a pretrained Bert model. Instead of fine-tuning all parameters of a model, it is common to directly reuse partial parameters and only fine-tune a small set of model layers for two reasons: (i) It is able to significantly decrease the training overheads of similar downstream tasks, enabling rapid development; (ii) Fine-tuning the whole model may result in overfitting if the target dataset is small and the number of parameters is huge [23, 28, 34, 44, 48]. It even improves the accuracy in case of insufficient training samples (e.g., sporadic noises). For example in the advertising business, a Resnet50 model with only the top layers retrained achieves an accuracy of > 98% in classifying QR codes.

It is highly empirical to determine which layers should be reused in practice. For tasks like sentiment classification, reusing the initial 12-16 layers of Bert even outperforms fine-tuning all layers on the SST-2 dataset [39]. The VGGish-based audio classification model for detecting noises with retained bottom embedding layers also achieves an accuracy of > 93%. Table 1 summarizes the representative deep learning models used for image, text, and audio processing in our website, as well as their applications, potential redundant parts and redundancy ratios.

Table 1: Tensor redundancy at 58.com.

Application Domains	Image	Text	Audio
Representative Models	Resnet50	Bert	
	EfficientNet	Roberta	VGGish
	MobileNet	Albert	
		GPT	
Applications	Advertising	Reading Extraction	Audio Classification
	Housing	Text Summary	
	Secondhand Trading	Text Classification	
Redundant Part	Bottom Layers	Embeddings Middle Layers	Embeddings
Redundancy Ratio	>70%	>31%	>90%

Observation #5: Cache redundancy: *Both the runtime and tensor redundancy are exacerbated by the widespread usage of in-memory caches in both the serverless platform and DL library.*

To alleviate the cold-start overhead, serverless platforms [20, 60] often keep function alive and warm after the execution

is finished (a.k.a. caching). Caching exacerbates the runtime and tensor redundancy problem.

Moreover, caching also exists in DL frameworks. For example, TensorFlow makes use of high-performance computing libraries, such as MKL-DNN [32] to accelerate the execution of computational graphs. The internal tensor representation of TensorFlow is not identical to MKL-DNN when performing operations such as convolutions; thus, the convolutional kernel data in TensorFlow format must be converted into MKL-DNN format. These converted parameters are usually cached in memory for subsequent usage [4] (e.g., 87.7% of parameters are cached for Resnet50). Hence, these caches also increase tensor redundancy.

2.3 Implications

Due to the memory-intensive nature of startup and computing in DL inference (**Observation 1** and **Observation 2**), the main memory can be easily and massively harvested in serverless inference systems. Reducing the memory footprint of serverless inference needs to be addressed immediately. We study the in-memory data redundancy in serverless inference systems, and observe that the redundancy problem primarily derives from three aspects: *runtime redundancy* (**Observation 3**), *tensor redundancy* (**Observation 4**) and *cache redundancy* (**Observation 5**).

Table 2: Comparison of existing systems.

	KSM [33]	Photons [16]	INFless [71]	TETRIS
Runtime sharing	✓	concurrency	batching	✓
Tensor sharing	✓	✗	✗	✓
Func. Cache sharing	✓	✗	✗	✓
No-harming perf.	✓	limited	limited	✓
First-time sharing	✗	✓	✓	✓
N.F. Running level	kernel	container	container	container
Imple. difficulty	high	low	low	low

Prior works [3, 16, 33, 47, 71] have proposed reducing the in-memory data redundancy through *page merging* or *runtime sharing* (Table 2). The *page merging* methods, such as KSM [3, 33, 47], support memory-saving deduplication through searching and merging equal physical pages of memory. However, it is a Linux kernel-level feature; therefore, its implementation is relatively difficult. It also does not support *first-time sharing* during the startup of function instances. *Runtime sharing* can be enabled through either *concurrent execution* (e.g., Photons [16]) or *batching* (e.g., INFless [71]). However, they only partially solve the *runtime redundancy* problem and their optimal combination is still worth further exploration. Moreover, the *tensor redundancy* and *cache redundancy* still exist and severely degrade the memory efficiency.

3 System Design

In this section, we present the design of TETRIS.

3.1 System Overview

The insight of TETRIS is that memory efficiency can be improved through the combined optimization of tensor

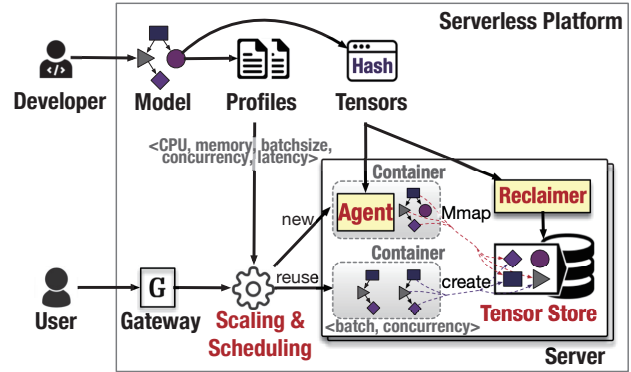


Figure 6: An overview of TETRIS.

sharing and runtime sharing. Figure 6 illustrates the overall architecture of TETRIS, which enables runtime sharing through its *scaling and scheduling* engine and supports tensor sharing through the *agent* in each container and the *tensor store* in each server. When a developer submits a trained DL model to the platform, TETRIS extracts the tensor information (including tensor sizes and hash values) and profiles the inference latency under various batch sizes and concurrency settings. The tensor information is used for directing *tensor sharing*, and the profiles are used by the *scaling and scheduling* module to determine the best *runtime sharing* configuration.

When the user submits requests to the *gateway*, the *scaling and scheduling* engine first decides whether to scale out new instances or reuse existing ones according to the workloads. First, in the case of scaling out instances, the engine derives the batch size and concurrency configurations, minimizing the memory footprint under the SLO guarantee. Then, the new instances are launched on servers with maximum *tensor similarity*. TETRIS supports *first-time sharing* of tensors through a special *agent* in each sandbox. The *agent* parses the computational graph of the model and reads the *hash* value of the tensors. Then, the agent checks whether the *tensor store* has already stored it. If so, the *agent* only maps the memory address of the tensor to its local process using the syscall of *Mmap*, and the reference number of the tensor is increased by 1. Otherwise, the agent loads the tensor from the model file into memory directly and creates a new item in the *tensor store*. Each *agent* is only active during the startup of a new instance. In contrast to DL training where tensors may be updated every iteration, tensors in inference are fixed during its service time. Hence, we set them to be only readable to ensure safe memory access across functions. Second, in the case of reusing existing instances, requests are forwarded to instances following the load balancing principle.

When the workload decreases, the *scaling and scheduling* engine also selectively releases the least-loaded instances without violating the SLO. After release, the reference number of its tensors decreases by 1. In each server, there is a *memory reclaimer* periodically validates the reference number of each

tensor and reclaims the memory pages with reference = 0 so that the released functions no longer occupy memory. A *keep-alive* policy can be adopted to avoid reclaiming oscillation.

3.2 Scaling and Scheduling

The *scaling and scheduling* engine relies on the model profile and requests per second (RPS) to make the scaling decision.

Profiling: We develop an automatic profiler that measures the inference latency of each model under various configurations. Specifically, we define the profile of each model as a 5-tuple $\langle c, m, b, p, l \rangle$, where $c \in C$ denotes the number of allocated CPUs, $m \in M$ denotes the memory configuration for the inference, $b \in B$ indicates the maximum batch size for that instance to process requests, $p \in P$ represents the number of concurrent inference threads, and l represents the inference latency under previous configurations. Since current serverless platforms typically use CPUs for function computation and DL inference on CPUs typically uses small batches (e.g., $B = \{1, 2, 4\}$) and low concurrency (e.g., $P = \{1, 2, 3, 4\}$) to achieve low latency. Thus, we profile only these combinations to narrow the profiling space. Since allocating excessive memory does not lower inference latency, we assign each configuration the bare minimum. Finally, we obtain $n = |C||M||B||P|$ 5-tuples characterizing the model, which are stored in the database.

In the profiling phase, TETRIS also computes and stores the hashes of tensors using the cyclic redundancy check (CRC) code, which is utilized in checking the memory status during *tensor sharing*.

Runtime-shared scaling: The scaling engine monitors the real-time RPS and judges whether the existing instances are sufficient to serve these requests. If not, it dispatches parts of requests to existing instances and launches new instances to process the residual ones. Given the model profile and residual RPS (denoted by R), the scaling engine explores the optimal configurations of new instances, to minimize memory usage while guaranteeing their latency SLO. We define an integer variable $x_i, \forall i \in [1, \dots, n]$, which indicates the configuration i is adopted for x_i new instances. Hence, the optimal configuration can be found by solving the following integer programming problem:

$$\text{minimize : } \sum_{i=1}^n m_i x_i \quad (1)$$

$$l_i \leq t_{slo}, \quad \forall i \wedge x_i \geq 1 \wedge b_i = 1 \quad (2)$$

$$l_i \leq t_{slo}/2, \quad \forall i \wedge x_i \geq 1 \wedge b_i > 1 \quad (3)$$

$$\sum_{i=1}^n x_i b_i p_i / l_i \geq R, \quad \forall i \quad (4)$$

$$x_i \in \mathbb{N} \quad (5)$$

Objective (1) defines the memory occupied by the instances. Constraints (2) and (3) ensure that the latency SLO is satisfied: For $b_i = 1$ (i.e., no batch queue exists for an inference thread), the inference latency l_i should not exceed the SLO; For $b_i > 1$ (i.e., requests must wait in a queue to saturate the batch), we have $t_{wait}^i + l_i \leq t_{slo}$, where t_{wait}^i denotes the waiting time in the batch queue. Suppose the RPS distributed to instance i is r_i , then we have $t_{wait}^i = b_i p_i / r_i$. Since $r_i \leq b_i p_i / l_i$ must be

Algorithm 1: DTS Algorithm

Input:

Requests R ; profile $O = \{ \langle c, m, b, p, l \rangle \}; t_{slo}$;

Output:

S : the set of selected instance configurations;

```

1  $S = \emptyset$ ;
2 sort  $O$  in descending order of  $(b_i p_i) / (l_i m_i), \forall i \in [1..n]$ ;
3 while  $R > 0$  do
4   for each configuration  $o_i \in O$  do
5     if  $b_i = 1 \wedge t_{exec}(c_i, b_i, p_i) > t_{slo}$  then
6        $\lfloor$  continue;
7     if  $b_i > 1 \wedge t_{exec}(c_i, b_i, p_i) > t_{slo}/2$  then
8        $\lfloor$  continue;
9      $R \leftarrow R - (b_i p_i) / l_i$ ;
10     $S \leftarrow S \cup \{o_i\}$ ;
11    break;
```

established (otherwise, if the request arrival rate exceeds the batch processing rate, requests will be dropped), we obtain $l_i \leq t_{slo}/2$. Constraint (4) ensures that the residual RPS can be fully processed by the new instances. Constraint (5) refers to the domain constraint.

This problem can be reduced to the NP-Complete knapsack problem [54]. Hence, we design a heuristic algorithm, called Decreased Throughput Selection (DTS) to determine the instance configuration efficiently. Algorithm 1 presents the details. In line 2, we sort the configurations in O in descending order by normalized throughput. Then, we greedily select the configuration with higher normalized throughput as long as the latency SLO is satisfied (lines 3-11). The DTS algorithm can also be easily extended to balance CPU usage by normalizing the throughput with $m_i + \alpha c_i$, where α denotes the equilibrium factor.

Scheduling: Once the scaling engine has determined the configurations of new instances, the scheduler deploys them to the appropriate servers. To reduce cluster-level memory consumption through *tensor sharing*, TETRIS always tries to dispatch them on servers with maximum *tensor similarity* (denoted by θ). Specifically, for instance i , TETRIS first filters out servers that do not meet resource requirements (e.g., CPU and memory). Then, it derives the similarity value between a model i and a server j using $\theta_{ij} = Mem(T_i \cap T_{store}^j) / Mem(T_i)$, where T_i represents the set of tensors in instance i , T_{store}^j represents the set of tensors already stored in server j , and $Mem(T_i)$ represents the aggregated memory of tensors in T_i .

3.3 Tensor Sharing

After an instance is scheduled on a server, TETRIS first launches a sandbox (e.g., container) on the target server to accommodate it. Then, an *agent* is activated in the sandbox to load the model into the main memory. For a tensor that has

previously been loaded into the *tensor store*, the *agent* maps its memory address to the instance. Otherwise, it creates a new item for it in the *tensor store*.

```

1  Status LoadTensor(Tensor& tensor, TensorReader& reader) {
2      // Get tensor hash value.
3      std::string tensor_hash = GetHash(reader, tensor);
4      // Get or create tensor lock in
5      // Shared Tensor Store atomically.
6      TensorLock lock = CreateOrGetTensorLock(tensor_hash);
7      // Obtain ownership of a tensor lock.
8      lock.Lock();
9      // Check if the tensor in Shared
10     // Tensor Store already exists.
11     if(!TensorExists(tensor_hash)) {
12         // Allocate the tensor memory in
13         // Shared Tensor Store and load
14         // the model parameters.
15         CreateTensor(reader, tensor, tensor_hash);
16     } else {
17         // Mapping already existing tensor
18         // memory from the Shared Tensor Store.
19         MmapTensor(tensor, tensor_hash);
20     }
21     // Release the lock.
22     lock.Unlock();
23     return Status::OK();
24 }

```

Listing 1: Simplified code snippet for loading tensors.

Agent: The agent is a lightweight, user-space process for sharing tensors across function instances. Whenever loading a model into memory, the *agent* reads the computational graph metadata stored in the model file and parses the tensors that need to be loaded into memory. As illustrated in Listing 1, the *agent* reads the hash value of a tensor using the interface *GetHash()* and checks whether the *tensor store* has already had the tensor. If the *TensorExists()* returns *FALSE*, the *agent* allocates memory for the tensor and puts it in the *tensor store*. Otherwise, it calls *MmapTensor()* to map the memory address of the existing tensor to the model. The *agent* maintains a loading queue for tensors. To ensure that the *tensor store* correctly behaves when it is accessed by multiple concurrent *agents*, we employ locks: an *agent* must acquire a lock before writing to the store (line 8) and release it after writing completion (line 22).

The *agent* is only active during the startup of an instance. It does not require modifications to the underlying operating system or virtualization layer and does not degrade inference performance.

Tensor store: The *tensor store* holds all tensor memory (parameters, constants, etc.) that need to be shared across all function instances on a server. Every function instance on the server can access tensors in the *tensor store*. Each tensor is uniquely identified by a hash value, calculated from the tensor content and dimensions. We use hash values because they are independent of the models and underlying frameworks. There is also a corresponding lock for each tensor to ensure their safe operation of constructing, mapping, or reclaiming. The *tensor store* is initially empty and does not hold any tensors or locks. During the running of the system, the *agent* continuously adds tensors into it. Each tensor is associated with a *reference number* initialized with 1 after its creation.

Whenever the *agent* adds a new mapping to an existing tensor, the *reference number* is increased by 1. Similarly, whenever an instance is released after completion, the *reference number* is decreased by 1. The tensor memory is reclaimed after the *reference number* is set to 0.

While the *tensor store* is shared by all function instances by default, TETRIS also supports building a *dedicated tensor store* for a subset of functions at the local server (e.g., functions belonging to the same tenant). The tensors in a *dedicated tensor store* cannot be accessed by functions without permission.

3.4 Memory Reclaiming

When a function instance is released, the tensor that it maps from the *tensor store* is not instantly deleted, as it may still be referenced by other instances. To appropriately reclaim the memory of shared tensors, TETRIS runs a *memory reclaimer* on each server, which periodically detects and reclaims the memory of tensors with *reference number* = 0.

The *reclaimer* also supports *tensor caching* policies, which keep tensors in the *tensor store* even after their *reference numbers* become 0. Currently, we have added two caching policies in the *reclaimer*: (1) *keep-alive window*: it is a timeout threshold to determine how long a tensor is kept alive; (2) *Least Recent Used (LRU)*: it only keeps the recently or often-used tensors in the *tensor store* after the *tensor store* is full. The cached tensors can accelerate the startup of function instances. Although TETRIS only supports these two policies at this time, other caching policies [20, 60] can be easily integrated into it. When a tensor’s memory is reclaimed, the *reclaimer* is also required to obtain its lock. To sum it up, Figure 7 depicts the lifecycle of tensors in TETRIS.

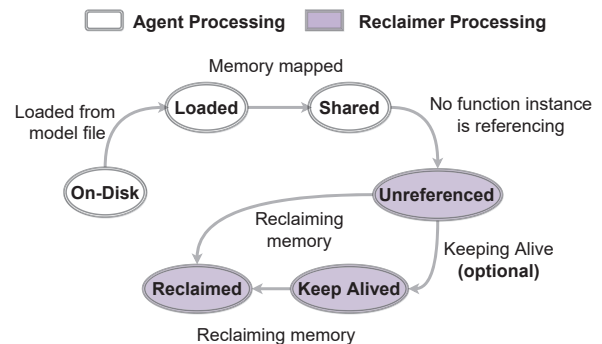


Figure 7: Overview of the tensor lifecycle in TETRIS.

4 System Implementation and Discussion

TETRIS is implemented in the open-source serverless platform OpenFaaS [17] with the DL inference framework TensorFlow Serving [22]. In particular, the *runtime sharing* ability is added to OpenFaaS by modifying its modules including *request dispatching*, *autoscaling*, *scheduling* and *instance creating*; The *tensor sharing* ability is implemented by modifying the TensorFlow Serving framework. We further implement

the *reclaimer* as a separate daemon. The entire system runs on Kubernetes. Overall, TETRIS’s implementation introduces negligible pollution to the existing software stack except for the newly added module *reclaimer*.

Tensor store: Since maintaining a cluster-level global tensor store is costly due to frequent tensor access during inference and high network latency, TETRIS maintains a shared tensor store on each server for performance guarantee while minimizing cluster memory consumption through instance scheduling. The tensor store on each server is implemented as a shared memory region, which can be accessed by all *agents* and the *reclaimer* at the local server. Although shared memory can be enabled by Docker through setting the `-ipc=host` option at the container creation time, allowing all containers to share the host *ipc* namespace, this introduces significant risks of malicious activities or misoperations. Hence, we instead implement the shared memory by mounting a memory-based *tmpfs* [64], in which the tensor store is just a directory. Then, it could be mounted to each container during its creation time using command like `docker -v`. Tensors are stored as files under the mounted *tmpfs* directory and their hash values are set as the filenames. In this way, we can build multiple dedicated tensor stores flexibly, just by creating different mounted directories.

Agent: The *agent* is integrated into the existing loading process of the TensorFlow Serving system. In particular, we modify the *RestoreOp* interface and provide a new tensor memory allocator to manage the shared memory mappings using the *open* and *Mmap* syscalls. After a tensor is created and initialized, its memory pages are tagged as read-only to prevent modifications. File locks are leveraged to synchronize *agents*, avoiding manipulating tensors simultaneously. In the case of concurrently loading models, we pre-randomize the list of loaded tensors to reduce lock conflicts. Besides, we replace the *malloc* interface in the TensorFlow Serving framework with *tcmalloc* [59] to reduce the memory waste.

Scaling & scheduling: The scaling and scheduling engine is integrated into the *faas-netes* module of OpenFaaS. In particular, we modify its *autoscaling* module to implement the DTS algorithm. While OpenFaaS originally uses the default scheduler of Kubernetes, we also modify it using the *tensor similarity*-based scheduling algorithm. To enable both *runtime sharing* and *tensor sharing*, we directly create Kubernetes pods for instances and mount the *tmpfs* directory in the instance creating process.

Reclaimer: The *Reclaimer* is implemented as a Kubernetes DaemonSet. Although Linux provides *fuser* or *lsof* for identifying whether or not a tensor has been referenced by processes, these tools are highly inefficient. Instead, the *Reclaimer* retrieves the set of running functions on a server through Kubernetes’ API, then infers the set of referencing tensors (denoted by T_{warm}) by querying the profile database. Then, the tensor set to be reclaimed (denoted by T_{cold}) can be derived using $T_{cold} = T_{all} \setminus T_{warm}$, where T_{all} represents the set of tensors

residing in the Tensor Store. Such an implementation also ensures fault tolerance, i.e., unreferenced tensors can be detected even when *agents* crash.

GPU Inference: Although the current serverless platforms typically use CPUs for function computation, TETRIS is still effective for GPU inference since frameworks like TensorFlow usually keep a copy of tensors in CPU memory. TETRIS is also suitable for large models which cannot fit into GPU memory. As GPU memory could also be shared through *cudaIpc* [50] APIs, TETRIS can be extended to the GPU inference scenario by developing a specialized manager for maintaining GPU tensor mappings.

5 Evaluation

5.1 Methodology

Table 3: Experimental testbed configurations.

Machine Type	Type 1	Type 2
CPU Device	Intel(R) Xeon(R) CPU E7-4820 v4	Intel Xeon Silver-4215
Number of Sockets	4	2
Processor BaseFreq	2.0 GHz	2.50 GHz
CPU Threads	80 (40 physical cores)	32 (16 physical cores)
Memory Capacity	256 GB	128 GB
Shared LLC Size	25 MB	11 MB
Operating System	Ubuntu 16.04	Ubuntu 16.04
Kubernetes Version	v1.19.2	v1.20.0

Testbed: We evaluate TETRIS using an 8-server testbed consisting of two types of machines: 2 machines are equipped with 80 cores and 256GB of memory while 6 others are equipped with 32 cores and 128GB memory. The machines are interconnected via a 100 Gbps, full-bisection bandwidth Ethernet. Table 3 lists the hardware and software details.

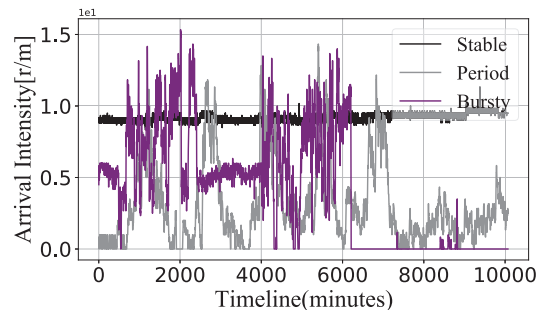
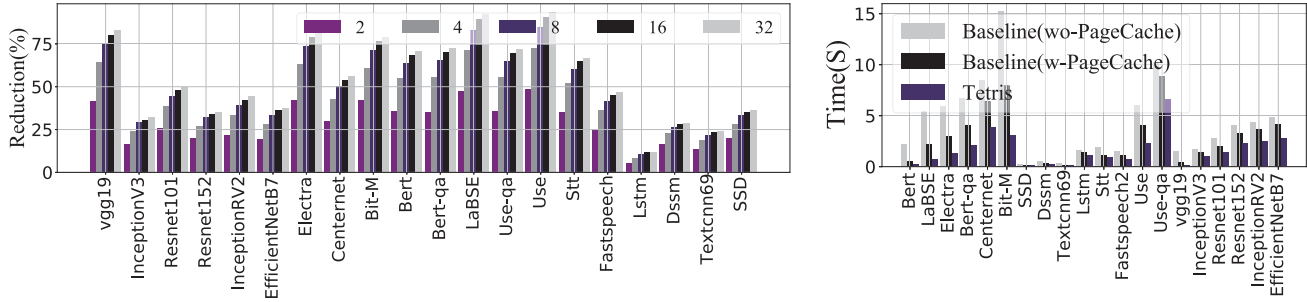


Figure 8: Production workload trace examples.

Workloads: We collect a comprehensive set of benchmark DL models with high heterogeneity from both TensorFlow Hub [22] and the real-world local life service website (Table 4). The benchmark suite is comprised of 21 inference models, ranging in model sizes from 11MB to 3.5GB, varying in download times from 310 to 1.1M, and covering application domains including text, image, audio, etc. Based on these models, we further construct four real-life applications: A *second-hand vehicle trading (SVT) application* adopts the SSD model for object detection and two Resnet152 models for classifying cars and motorcycles. An *audio question & answering (QA-Audio) application* employs the Stt, Use-qa and Fastspeech2 models for speech-to-text translation, QA



(a) Memory reduction under different # of instances

(b) Intra-model acceleration

Figure 9: (a) Memory reduction rate under tensor sharing over the number of instances from the same function. (b) Accelerating the startup using tensors from existing instance of the same function.

Table 4: Inference benchmark suite.

DL Model	Size	Description	Download times
Bit-M [37]	3.5GB	Feature vector extraction	1.4k
LaBSE [19]	1.8GB	Sentence Embedding	24.9K
Bert-qa [14]	1.3GB	Question Answering	501
Electra [10]	1.3GB	Discriminator	6.1K
Use [7]	980MB	Sentence Encoder	1.4M
Centernet [75]	731MB	Object Detection	12.8K
Use-qa [72]	568MB	Question Answering	16.3K
Use-large [7]	563MB	Sentence Encoder	1.1M
Vgg19 [63]	549MB	Image Classification	commercial
Bert [14]	392MB	Text Processing	197.5K
EfficientNetB7 [68]	255MB	Image Processing	2.2K
Resnet152 [26]	231MB	Image Processing	1.6K
InceptionResnetV2 [66]	214MB	Image Processing	6K
Stt [69]	176MB	Speech-To-Text	398
Resnet101 [26]	171MB	Image Processing	1.6K
FastSpeech2 [57]	119MB	Text-To-Speech	310
InceptionV3 [67]	92MB	Image Processing	11.6K
SSD [43]	29MB	Object Detection	commercial
Dssm [29]	25MB	Text Processing	commercial
Lstm [27]	23MB	Text Processing	commercial
Textcnn69 [9]	11MB	Text Processing	commercial

and text-to-speech translation, respectively; A *semantic similarity computation (SS) application* uses the *Use* model for semantic similarity computation; A *text question & answering (QA-Text) application* uses Textcnn69, Lstm, and Dssm for understanding user questions and finding matched answers.

These services are triggered by dynamic invocations simulated using the production trace from the Azure Function [46], where the invocations per hour illustrate diurnal and weekly patterns. Figure 8 displays all the three typical types of production traces used: stable, periodic, and bursty.

Competing approaches: We compare TETRIS with *INFless* [71], *Photons* [16], and the runtime sharing-only version of TETRIS: *Tetris-RO*. *INFless* is a state-of-the-art serverless inference system that natively supports batching and fine-grained CPU-GPU allocation for low-latency, high-throughput inference on serverless. *Photons* supports runtime sharing through concurrent execution in each instance. Since *Photons* is not serverless inference oriented and provide no SLO guarantee, we redevelop it atop OpenFaaS and extend it with function profiles while greedily selecting instance

configurations with the maximum concurrency under SLO constraints. *Tetris-RO* is a variant of TETRIS that disables the tensor sharing part.

5.2 Tensor Sharing Evaluation

We first evaluate the tensor sharing effectiveness by solely activating the TETRIS’s *agent*, and compare the memory footprint to that in the native OpenFaaS system. In particular, the memory footprint under tensor sharing can be derived by $M_{ts} = M_{tensor} + I \times M_{others}$, where M_{tensor} denotes the memory for parameterized tensors, M_{others} indicates the memory for runtime, libraries and others that cannot be shared across instances, and I represents the number of colocated instances in a server. Likewise, the memory footprint under the native OpenFaaS system can be derived by $M_{baseline} = I \times (M_{tensor} + M_{others})$.

Memory footprint: Sharing tensors across instances of an inference function saves memory by up to 93%. Figure 9(a) depicts the memory reduction rate by various models under various number of instances (from 2 to 32). Clearly, the memory reduction rate benefits more from increasing the number of colocated instances. The reduction rate depends on the percentage of memory that can be shared. Basically, the more parameters a model contains (i.e., M_{tensor}) and the less other memory (i.e., M_{others}) a model requires lead to a higher memory reduction rate. For example, the memory reduction rate of VGG19 (i.e., with $M_{tensor} = 549MB$ and $M_{others} = 95.4MB$) reaches to 82% when colocating 32 instances. For models with fewer parameters and relatively larger temporary memory footprints, Lstm for example, the memory reduction rate is limited to 4.4% in the 2-instance co-locating scenario. When deploying 32 function instances, the memory reduction rates of large models like LaBSE and Use even exceed 91% and 93%, respectively. For the model of Bit-M, TETRIS reduces its memory consumption by 108.5GB, because TETRIS only keeps a single copy of the model parameters in memory. Even with only two instances, the memory footprint can still be reduced by an average of 28%.

Memory footprint: Sharing tensors across functions can further reduce memory by up to 36.3%. We collect model variants which are generated by transfer learning from the InceptionV3, EfficientNetB7, Resnet101, Resnet152, Vgg19,

and InceptionResnetV2 models for applications in housing rental, recruitment, second-hand products, travel, and catering businesses. They are commonly only retrained the top dense layers. Figure 11(a) reveals that, as the number of model variants increases, the system memory can be further reduced for all models by up to 36.3%, and at least by 18.8%. Among these models, the memory reduction rate for Vgg19 is relatively limited since merely 77MB parameters are shared among its variants, although it still achieves 13.8% under the co-location of 32 variants.

Higher function density: Tensor sharing reduces the memory consumption of functions, improving the function density by up to 30 \times . Figure 10 presents the increasing rate of function density under various memory configurations. For the LaBSE model with 1.8GB parameters that consume 1.9GB memory, the function density is improved by 20 \times . In the case of the Use model, it even achieves an improvement of 30 \times . For large models, such as Bit-M, a server with 64GB of memory is only capable of maintaining 14 instances. After sharing tensors, the density increases to 74, i.e., more than 60 instances can be accommodated in the same machine. Tensor sharing still significantly improves function deployment density for models with small memory footprints. For instance, 1161 and 1720 additional instances can be created on a 256GB-memory server for Dssm and Textcnn69, respectively.

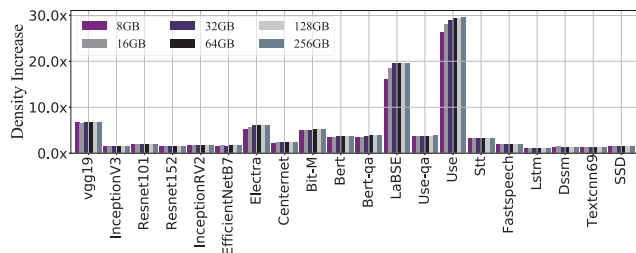
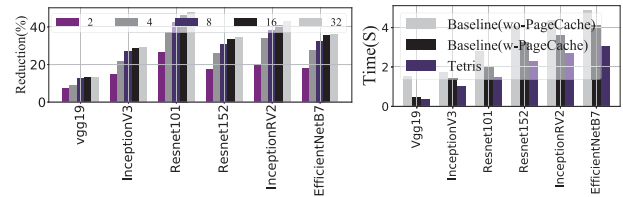


Figure 10: Function density improvement under various machine memory capacities.

Accelerating startup: The first-time sharing of tensors reduces the startup overhead of inference functions, accelerating the startup speed by 91.56%. The loading of tensors dominates the startup process. Directly mapping an existing tensor address to a new instance is much faster than loading and decoding it from the file on disk; thus, the first-time tensor sharing feature of TETRIS can significantly accelerate the startup process. Figure 9(b) demonstrates that tensor sharing from the same function’s previous instance can significantly speed up the startup process. For example, the model loading time of Bit-M exceeds 15 seconds in a native system whose page cache is disabled, which remains to be 7.9s with page cache acceleration, while tensor sharing can reduce its startup time to 2.8 seconds. Even for models with fewer tensors, such as SSD and Textcnn69, the speedup from tensor sharing still achieves 17.3% and 32.8%, respectively.

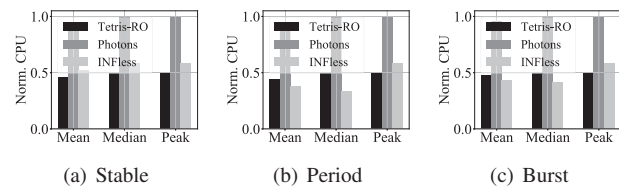
Besides the acceleration from sharing tensors of the same function, cross-function tensor sharing could accelerate

startup even when the model has never been loaded. Although it may be slower than that from the same function’s instance due to less redundancy across functions, the speedup still achieves an average of 46.9% (Figure 11(b)). For Vgg19, the startup process is still accelerated by 12.3% when compared with the page cache.



(a) Memory reduction across functions (b) Inter-model acceleration

Figure 11: (a) Memory reduction rate under tensor sharing over the number of model variants. (b) Accelerating the startup using tensors from the existing instance of a different function.



(a) Stable (b) Period (c) Burst

Figure 12: CPU consumption of the SS application under varying workloads.

5.3 Overall Evaluation

We further evaluate the overall efficiency of TETRIS by deploying four applications: *SVT*, *QA-Audio*, *SS* and *QA-Text*, and requests towards them are generated using three types of production traces in Figure 8.

Memory footprint: TETRIS outperforms both *INFless* and *Photons* significantly in memory consumption. Figure 13 presents the normalized mean, median and peak memory consumption by TETRIS, *INFless* and *Photons*. We find that *INFless* consumes most of the memory in all experiments for two reasons: (1) *INFless* can only reduce memory consumption when the model supports batching. However, for the applications *QA-Audio* and *SS*, there are models (i.e., *FastSpeech2* and *Use*) that do not support batching. (2) Batching introduces additional queuing time. While the inference computation on CPU is slow and the SLO is tight, we are not able to configure a much larger batch size even for batch-enabled models. (3) *INFless* prefers to use the fragmented resources spanning over multiple servers for better accommodating the residual load and improving the resource utilization, which exacerbates memory consumption. For the four applications *SVT*, *QA-Audio*, *SS* and *QA-Text* under a *stable* request load, TETRIS can reduce the mean memory footprint by more than 86%, 64%, 69%, 65%, respectively, and reduce the peak memory consumption by more than 87%, 68%, 71% and 65%, respectively. In particular, TETRIS achieves the best memory

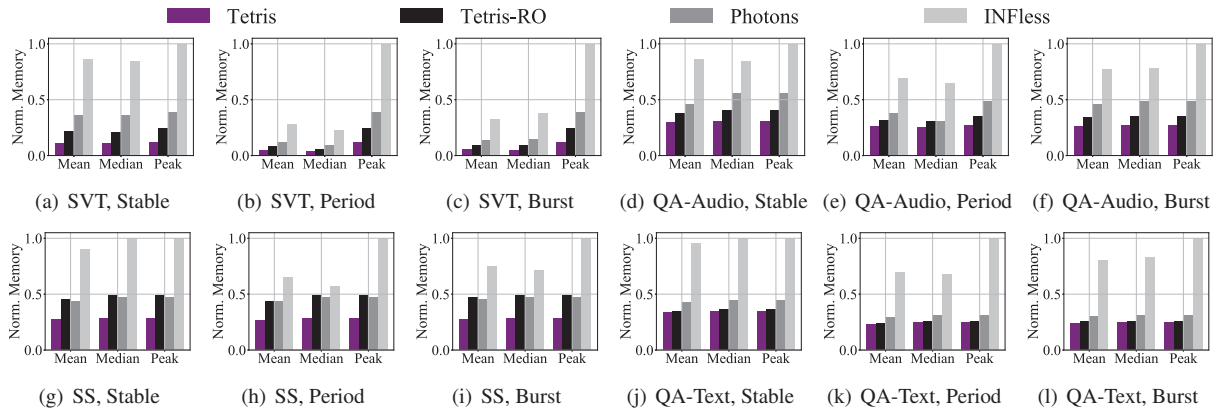


Figure 13: Normalized memory consumption by four applications under stable, period and bursty workloads.

efficiency in the SVT case because its Resnet152 models benefit more from tensor sharing as they are built from the same set of pretrained parameters. Resnet152 is also computationally intensive and limits the chance for packing requests to share runtimes. *Photons* consumes much less memory than *INFless* since it executes requests concurrently within the same instance without batch queuing time, however tensor redundancy across function instances still exists. Because of tensor sharing, TETRIS can further decrease the mean memory consumption by more than 68%, 33%, 37%, and 21%, and the peak memory consumption by more than 67%, 43%, 39%, and 22% for the applications, respectively.

Efficient runtime sharing: The combined optimization of batching and concurrent execution in TETRIS outperforms either *INFless*'s batching or *Photons*'s concurrent execution. As illustrated in Figure 13, *Tetrtris-RO* can reduce more memory footprint than *INFless* and *Photons* in cases of SVT, QA-Audio, and QA-Text: *Tetrtris-RO* can select configurations flexibly by exploring the various combinations of batch size, concurrency and CPUs. Taking SVT as an example, *Photons* only chooses to execute at most 2 concurrent requests within each instance due to its fixed mapping between concurrency and CPUs, while *Tetrtris-RO* can greedily activate 3 concurrent threads. For the QA-Audio application, although *Photons* greedily packs 4 requests into the same instance, *Tetrtris-RO* could further decrease runtime redundancy by concurrently executing 2 requests with a batch size of 6. For the QA-Text application consisting only of small models, the average memory consumption is still reduced by 21% from the runtime sharing.

Although *Tetrtris-RO* consumes more memory than *Photons* in the SS application case, *Tetrtris-RO* requires much fewer CPU resources. Figure 12 illustrates that *Tetrtris-RO* and *INFless* allocate CPU resources averagely by 42% and 39% less than *Photons*. This outcome is because each time *Photons* increases concurrency, it also requires a fixed amount of additional CPUs, resulting in excessive CPU allocations.

SLO guarantee: TETRIS can guarantee the latency SLO of inference workloads. Figure 14(a) demonstrates that both TETRIS and *INFless* achieve a low SLO violation rate (< 4%)

for all applications. For SVT and SS, TETRIS outperforms *INFless* since more CPUs are allocated for each instance in TETRIS to support concurrent processing. For QA-Audio and QA-Text, the SLO violation rate of TETRIS is slightly higher than that of *INFless* since it uses a larger batch size in runtime sharing, introducing additional batch queuing time. Overall, TETRIS achieves significant improvement in memory efficiency at the cost of a negligible increase in the SLO violation rate.

Cost savings: The reduction of memory consumption by TETRIS saves considerable monetary cost for inference service providers. To further demonstrate the benefits of TETRIS, we conduct a large-scale simulation with a combination of the mentioned applications, and gradually increase the RPS from 100 to 5000. As illustrated in Figure 14(b), TETRIS reduces memory of 61.5GB and 20.2GB per 100 requests, compared to that of *INFless* and *Photons*, respectively. If following the pricing model of \$0.0504 per hour according to the r6g.medium service at AWS EC2, such memory reduction can be transformed into \$0.000861 and \$0.000283 cost savings per 100 requests. If considering the local life website, which serves 1.9 billion requests per day, TETRIS could reduce the monetary cost by \$16,359 per day, about \$5.97 million per year.

5.4 Overhead

The implementation of TETRIS does not require modifications to the ML model, the virtualization sandbox, or the underlying operating system. Developers are simply required to submit the model (instead of the function code) as the model itself contains sufficient information for deployment.

Profiling overhead: TETRIS relies on the offline profiling phase to estimate latency under various CPU, batch size, and concurrency configurations. Since inference on CPU typically incurs high latency, the exploration space of combinations of batch size and concurrency is actually limited. The automatic profiler can test each inference model under various configurations and generate profiles within several minutes. In a real system, as inference services are invoked repeatedly, the profiling only generates a one-time cost and the profiles can

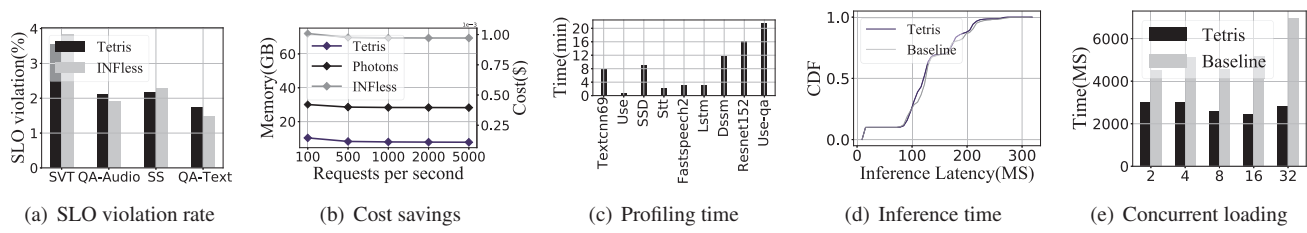


Figure 14: (a) SLO violation rate. (b) Memory consumption and financial cost per 100 requests under varying workloads. (c) Average profiling time for models used in the evaluation. (d) Inference latency distribution. (e) Average loading latency over the number of concurrent loading function instances with models from Table 4.

be reused for later invocations. As depicted in Figure 14(c), profiling an inference model takes an average of 12 minutes.

Inference latency: The memory address mapping method in TETRIS does not introduce latency overhead. We measure the latency distribution of the models in Table 4. Figure 14(d) demonstrates that, compared to the latency yielded by deploying on the TensorFlow Serving framework directly, there is no performance degradation observed after using TETRIS.

Lock contention: When multiple instances are created simultaneously, the contention on the tensor lock may cause startup overhead. We measure the average model loading latency without page cache and compare it with the baseline of the TensorFlow Serving framework. Figure 14(e) indicates that TETRIS still outperforms the baseline by employing randomization in the tensor loading process, thereby reducing much of the lock contentions. Once a tensor is loaded into memory successfully, TETRIS need only to map the memory address for newly launched instances, whereas the baseline still requires expensive file reading and decoding.

6 Related Work

Memory deduplication: Prior works [3, 8, 21, 47, 70] have proposed eliminating redundant data loaded in memory by scanning, comparing, and merging duplicated pages. However, such page-level scanning methods could incur colossal overhead and lag for large, high-density serverless inference functions. As they are effective at the kernel level, TETRIS is compatible with these methods and can be used in conjunction to reduce memory consumption further.

Model compression: Large inference models can also be substantially compressed through fine-grained model design, pruning and quantization techniques (e.g., [24, 25, 31, 56]). However, unlike TETRIS, these methods involve model modifications. The compressing process may also reduce inference accuracy, resulting in side effects for businesses [53]. TETRIS and model compression can coexist.

Inference runtime optimizations: The memory footprint of inference can also be reduced by optimizing memory allocation during inference runtime [13, 35, 40, 41, 55]. However, these optimizations are either framework-specific or require model conversion, whereas TETRIS is orthogonal to these optimizations and can be employed together to further reduce

memory consumption. Moreover, they may require developers to redesign or retrain the model, increasing the development burden, whereas TETRIS requires no model modifications.

Serverless inference: The existing serverless inference systems [1, 6, 71, 74] generally focus on improving inference throughput without violating the SLOs. Instead of improving memory efficiency, they primarily optimize the allocation of computational resources (e.g., CPU and GPU). TETRIS explores solving the memory efficiency problem in serverless inference and can be integrated into such systems. Trims [12] accelerates the data shipping between CPU and GPU through sharing existing models in GPU memory, whereas TETRIS explores sharing tensors among function instances.

7 Conclusion

Modern applications (such as IoT data processing, advertising recommendations, autonomous driving, and e-commerce) continuously rely on inference services. It is expected that serverless inference can reduce the maintenance cost for service providers, however, memory resources can be easily and massively harvested in existing serverless inference systems. Our proposal, TETRIS, can significantly reduce the memory footprint of inference services through runtime sharing and tensor sharing. TETRIS can be easily integrated into serverless platforms as a user-space plugin. With TETRIS, cloud providers can deploy an order of magnitude more instances in each server without violating the SLOs, thus significantly decreasing the deployment and maintenance costs. The prototype of TETRIS is available at <https://github.com/JelixLi/Tetris>.

In the future, we would like to further optimize the GPU memory efficiency of serverless inference systems.

Acknowledgments

We thank our shepherd and other anonymous ATC reviewers for their extremely insightful comments and suggestions that have significantly improved the quality of this paper. We also thank Dr. Tao Li for his significant contribution to this work. This work is supported by the National Natural Science Foundation of China under grant 61872265, 62141218, U1836214; the new Generation of Artificial Intelligence Science and Technology Major Project of Tianjin under grant 19ZXZNGX00010 and the open project of Zhejiang Lab (2021DA0AM01/003). It is also supported by Meituan.

References

- [1] Ahsan Ali, Riccardo Pincioli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.
- [2] Amazon. Aws lambda. <https://aws.amazon.com/lambda/>.
- [3] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. In *Proceedings of the linux symposium*, pages 19–28. Citeseer, 2009.
- [4] Mahmoud Abuzaina Ashraf Bhuiyan. Improving tensorflow* inference performance on intel® xeon® processors. <https://www.intel.com/content/www/us/en/artificial-intelligence/posts/improving-tensorflow-inference-performance-on-intel-xeon-processors.html>.
- [5] AWS. Serverless application lens: Alexa skills. <https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/alexaskills.html>. Referenced 2022.
- [6] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. Barista: Efficient and scalable serverless serving system for deep learning prediction services. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 23–33. IEEE, 2019.
- [7] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Céspedes, Steve Yuan, Chris Tar, et al. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*, 2018.
- [8] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. Cmd: Classification-based memory deduplication through page access characteristics. *ACM SIGPLAN Notices*, 49(7):65–76, 2014.
- [9] Yahui Chen. Convolutional neural network for sentence classification. Master’s thesis, University of Waterloo, 2015.
- [10] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.
- [11] Clive Cox, Dan Sun, Ellis Tarn, Animesh Singh, Rakesh Kelkar, and David Goodwin. Serverless inferencing on kubernetes. *arXiv preprint arXiv:2007.07366*, 2020.
- [12] Abdul Dakkak, Cheng Li, Simon Garcia De Gonzalo, Jinjun Xiong, and Wen-mei Hwu. Trims: Transparent and isolated model sharing for low latency deep learning inference in function-as-a-service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 372–382. IEEE, 2019.
- [13] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. Tensorflow lite micro: Embedded machine learning for tinymml systems. *Proceedings of Machine Learning and Systems*, 3:800–811, 2021.
- [14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [15] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *Proceedings of The 31st International Conference on Machine Learning*, pages 647–655, 2014.
- [16] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 45–59, 2020.
- [17] Alex Ellis. Serverless functions, made simple. <https://www.openfaas.com/>.
- [18] Fangxiaoyu Feng, Yinfei Yang, Daniel Cer, Naveen Arivazhagan, and Wei Wang. Language-agnostic bert sentence embedding, 2020.
- [19] Fangxiaoyu Feng, Yinfei Yang, Daniel Cer, Naveen Arivazhagan, and Wei Wang. Language-agnostic bert sentence embedding. *arXiv preprint arXiv:2007.01852*, 2020.
- [20] Alexander Fuerst and Prateek Sharma. Faas-cache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021.
- [21] Anshuj Garg, Debadatta Mishra, and Purushottam Kulkarni. Catalyst: Gpu-assisted rapid memory deduplication in virtualization environments. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 44–59, 2017.
- [22] Google. Tensorflow hub. <https://tensorflow.google.cn/hub/>.

- [23] Yunhui Guo, Honghui Shi, Abhishek Kumar, Kristen Grauman, Tajana Rosing, and Rogerio Feris. Spottune: transfer learning through adaptive fine-tuning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4805–4814, 2019.
- [24] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.
- [25] Song Han, Jeff Pool, John Tran, and William J Dally. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626*, 2015.
- [26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [27] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [28] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.
- [29] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 2333–2338, 2013.
- [30] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869*, 2014.
- [31] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and < 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.
- [32] Intel. Intel(r) math kernel library for deep neural networks (intel(r) mkl-dnn). <https://oneapi-src.github.io/oneDNN/v0/index.html>.
- [33] Hugh Dickins Izik Eidus. Kernel samepage merging. <https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html>.
- [34] Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Tuo Zhao. Smart: Robust and efficient fine-tuning for pre-trained natural language models through principled regularized optimization. *arXiv preprint arXiv:1911.03437*, 2019.
- [35] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, et al. Mnn: A universal and efficient inference engine. *Proceedings of Machine Learning and Systems*, 2:1–13, 2020.
- [36] Paresh Kharya and Ali Alvi. Using deepspeed and megatron to train megatron-turing nlg 530b, the world’s largest and most powerful generative language model. <https://developer.nvidia.com/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/>. Referenced 2022.
- [37] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big transfer (bit): General visual representation learning. In *Computer Vision—ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part V 16*, pages 491–507. Springer, 2020.
- [38] AWS Lambda. Netflix & aws lambda case study. <https://aws.amazon.com/cn/solutions/case-studies/netflix-and-aws-lambda/>.
- [39] Jaejun Lee, Raphael Tang, and Jimmy Lin. What would elsa do? freezing layers during transformer fine-tuning. *arXiv preprint arXiv:1911.03090*, 2019.
- [40] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. {PRETZEL}: Opening the black box of machine learning prediction serving systems. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 611–626, 2018.
- [41] Shuangfeng Li. Tensorflow lite: On-device machine learning framework. *Journal of Computer Research and Development*, 57(9):1839, 2020.
- [42] Jens Lindemann and Mathias Fischer. A memory-deduplication side-channel attack to detect applications in co-resident virtual machines. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 183–192, 2018.
- [43] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.
- [44] Pedro Marcelino. Transfer learning from pre-trained models. *Towards Data Science*, 10:23, 2018.

- [45] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.
- [46] Microsoft. Azure functions. <https://azure.microsoft.com/en-us/services/functions/>.
- [47] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. {XLH}: More effective memory deduplication scanners through cross-layer hints. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 279–290, 2013.
- [48] Sangwoo Mo, Minsu Cho, and Jinwoo Shin. Freeze the discriminator: a simple baseline for fine-tuning gans. *arXiv preprint arXiv:2002.10964*, 2020.
- [49] Philipp Muens. Serverless facebook messenger bot. <https://github.com/pmuens/serverless-facebook-messenger-bot>. Referenced 2022.
- [50] Nvidia. Cuda runtime application programming interface. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDA__DEVICE.html.
- [51] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.
- [52] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1717–1724, 2014.
- [53] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*, 2018.
- [54] David Pisinger and Paolo Toth. Knapsack problems. In *Handbook of combinatorial optimization*, pages 299–428. Springer, 1998.
- [55] Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens van der Maaten, and Kilian Q Weinberger. Memory-efficient implementation of densenets. *arXiv preprint arXiv:1707.06990*, 2017.
- [56] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.
- [57] Yi Ren, Chenxu Hu, Xu Tan, Tao Qin, Sheng Zhao, Zhou Zhao, and Tie-Yan Liu. FastSpeech 2: Fast and high-quality end-to-end text to speech. *arXiv preprint arXiv:2006.04558*, 2020.
- [58] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 753–767, 2022.
- [59] Paul Menage Sanjay Ghemawat. Tcmalloc : Thread-caching malloc. <http://goog-perftools.sourceforge.net/doc/tcmalloc.html>.
- [60] Mohammad Shahrads, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 205–218, 2020.
- [61] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014.
- [62] Karen Simonyan and Andrew Zisserman. Two-stream convolutional networks for action recognition in videos. *arXiv preprint arXiv:1406.2199*, 2014.
- [63] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [64] Peter Snyder. tmpfs: A virtual memory file system. In *Proceedings of the autumn 1990 EUUG Conference*, pages 241–248, 1990.
- [65] Vikram Sreekanti, Harikaran Subbaraj, Chenggang Wu, Joseph E Gonzalez, and Joseph M Hellerstein. Optimizing prediction serving on low-latency serverless dataflow. *arXiv preprint arXiv:2007.05832*, 2020.
- [66] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017.
- [67] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

- [68] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.
- [69] Alexander Veysov. Towards an imagenet moment for speech-to-text. *The Gradient*, 2020.
- [70] Nai Xia, Chen Tian, Yan Luo, Hang Liu, and Xiaoliang Wang. {UKSM}: Swift memory deduplication via hierarchical and adaptive memory region distilling. In *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, pages 325–340, 2018.
- [71] Yanan Yang, Laiping Zhao, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. INFless: a native serverless system for low-latency, high-throughput inference. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, Feb 28- March 4, 2022*, pages 1–14. ACM, 2022.
- [72] Yinfei Yang, Daniel Cer, Amin Ahmad, Mandy Guo, Jax Law, Noah Constant, Gustavo Hernandez Abrego, Steve Yuan, Chris Tar, Yun-Hsuan Sung, et al. Multilingual universal sentence encoder for semantic retrieval. *arXiv preprint arXiv:1907.04307*, 2019.
- [73] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *arXiv preprint arXiv:1411.1792*, 2014.
- [74] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 1049–1062, 2019.
- [75] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points. *arXiv preprint arXiv:1904.07850*, 2019.



PetS: A Unified Framework for Parameter-Efficient Transformers Serving

Zhe Zhou[†]
Peking University

Xuechao Wei
Peking University
Alibaba Group

Jiejing Zhang
Alibaba Group

Guangyu Sun*
Peking University

Abstract

Deploying large-scale Transformer models under the conventional *pre-train-then-fine-tune* paradigm is impractical for multi-task serving, because a full model copy for each downstream task must be maintained, quickly exhausting the storage budget. Recent algorithmic advances in Parameter-Efficient Transformers (PETs) have shown enormous potential to mitigate the storage overhead. They share the pre-trained model among tasks and only fine-tune a small portion of task-specific parameters. Unfortunately, existing serving systems neither have flexible PET task management mechanisms nor can efficiently serve queries to different tasks in batches. Therefore, we propose PetS, the first unified framework for multi-task PETs serving. Specifically, different PET tasks are expressed by a unified representation in the same framework, which enables flexible PET task management. Based on the unified representation, we design a specialized PET inference engine to batch different tasks' queries together and execute them with task-agnostic shared operators and task-specific PET operators. To further improve system throughput, we propose a coordinated batching strategy to deal with arbitrary input queries. We also develop a PET operator scheduling strategy to exploit parallelism between PET tasks. Comprehensive experiments on Edge/Desktop/Server GPUs demonstrate that PetS supports up to $26\times$ more concurrent tasks and improves the serving throughput by $1.53\times$ and $1.63\times$ on Desktop and Server GPUs, respectively.

1 Introduction

Recently, large-scale pre-trained Transformer models have revolutionized the field of artificial intelligence. Benefited from the practical *pre-train-then-fine-tune* paradigm, Transformer models such as Bert [9], GPT-2/3 [3, 45], Roberta [31], XLNet [59], T5 [46], and some other variants [28, 29] have achieved the leading-edge performance on various NLP (Natural-Language-Processing) tasks, including question-answering, sentiment-classification, text classification and machine translation, etc. Besides NLP tasks, some recent works also apply transformers to computer vision tasks [4, 11, 25, 32, 55, 61], which demonstrate comparable or even superior

[†]Work done during Zhe Zhou's internship at Alibaba DAMO Academy.

*Corresponding author.

performance against conventional Convolutional Neural Networks (CNNs). In brief, Transformers have been recognized as a milestone of artificial intelligence.

To date, a standard workflow has been shaped to apply Transformers to real-world applications. As is depicted in Figure 1, big companies like Google first pre-train the Transformer models like Bert [9] and GPT [3, 45] with large-scale datasets (Step ①). The unsupervised pre-training usually lasts for days to months, even trained on TPU clusters [3, 9]. The pre-trained models with rich task-agnostic knowledge are provided to application developers, who then fine-tune the pre-trained models on their private datasets in a supervised manner (Step ②). The fine-tuned task-specific models are finally deployed to cloud or edge servers (Step ④) to process different input queries. Such a workflow, however, is faced with the poor scalability issue in the pervasive multi-task serving scenarios [18, 19, 34, 40, 48, 53]. Since application developers fine-tune and maintain a full model copy for each downstream task, the storage overhead is proportional to the number of deployed tasks. Considering the enormous parameters (e.g., several hundred millions to several thousand millions of parameters) of Transformer models, the storage overhead will be huge. What is worse, conventional serving frameworks have to swap in and out models frequently if the GPU memory cannot hold all the invoked tasks, resulting in much lower serving throughput. Also, since the input queries are associated with different models, we cannot inference them in batches for higher serving throughput [7, 12, 16, 50].

Recent algorithmic advances in Parameter-Efficient Transformers (PETs) have shown enormous potential to solve these problems partially. They share the pre-trained model weights among tasks and only fine-tune a small portion of task-specific parameters for each downstream task [14, 20, 23, 24, 41, 62, 64]. By this means, the storage overhead is substantially mitigated, while the model accuracy is still comparable or even superior to the full-model fine-tuning counterparts. These methods, however, cannot run efficiently with existing Transformers serving frameworks [12, 37, 56]. On the one hand, due to the lack of PET task management mechanism and PET-oriented inference engine, we have to merge PET parameters into the shared model and still send full model copies to the frame-

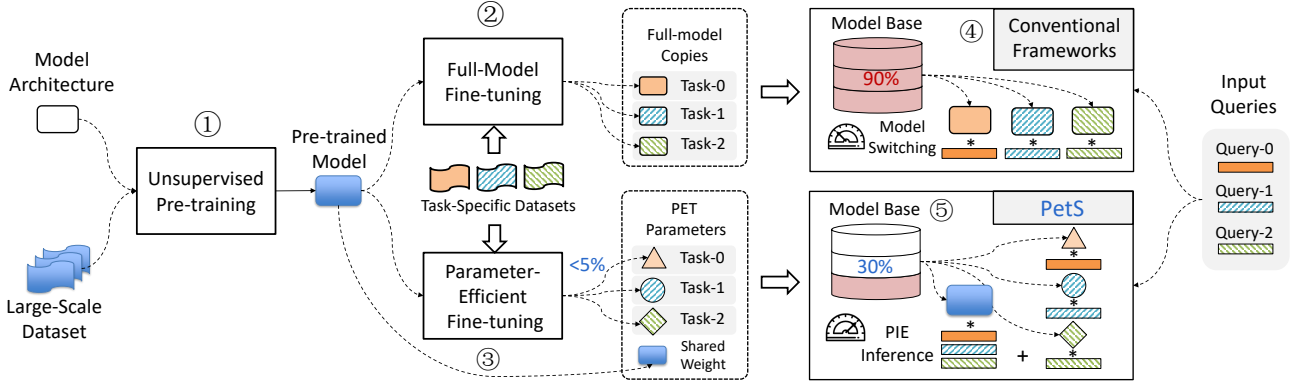


Figure 1: The conventional workflow VS. *PetS* workflow for developing and deploying Transformer-based applications. ①: Large-scale unsupervised pre-training. ②: Full-model fine-tuning on customized datasets for different tasks. ③: Parameter-efficient fine-tuning. ④: Serving the input queries with conventional frameworks. ⑤: Serving the input queries with *PetS*.

works. Thus, the GPU memory footprint is not mitigated. On the other hand, queries to different tasks cannot be processed in batches due to both the inter-task weight differences and inter-algorithm representation differences.

To take full advantage of parameter-efficient Transformers, in this paper, we propose *PetS*, a unified framework for multi-task PETs serving with extraordinary scalability and performance. To this end, we first express the state-of-the-art PET algorithms by a unified representation, which decouples any PETs into task-agnostic shared operations and task-specific PET operations. Based on the unified representation, we design a PET tasks management mechanism, which enables the service providers to register and load PET tasks flexibly. We then develop a high-performance PET Inference Engine (PIE) to batch different tasks' queries and execute them with shared operators and light-weighted PET operators, substantially improving the serving throughput. We also propose several optimization strategies to improve the system's throughput further. To be specific, we propose a Coordinated Batching strategy to deal with arbitrary input queries (i.e., queries with different sequence lengths and PET types). To exploit parallelism between PET operators, we apply a PET Operator Scheduling strategy to properly put concurrent PET operators to different CUDA streams. We comprehensively evaluate *PetS* on Edge/Desktop/Server GPU platforms. Compared to conventional frameworks, *PetS* supports up to $26\times$ more concurrent Transformer tasks and improves the serving throughput by $1.53\times$ and $1.63\times$ on Desktop and Server GPUs, respectively. Therefore, *PetS* shows great potential to reduce the service deployment cost and improve the service quality in multi-task Transformers serving scenarios.

2 Background & Motivations

2.1 Transformer Models

As illustrated in Figure 2, Transformer models are generally built by stacking several homogeneous Transformer blocks. A standard Transformer block consists of three key components:

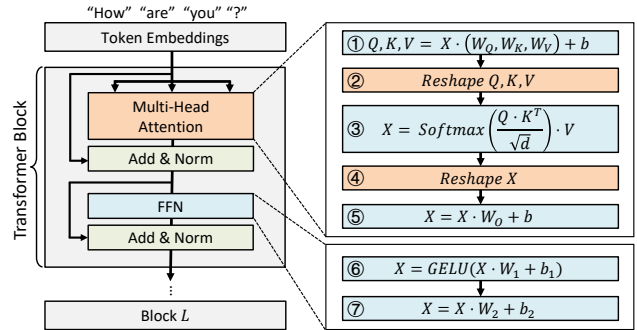
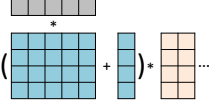
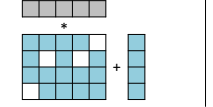
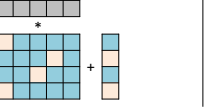
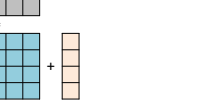


Figure 2: Bert Architecture.

Multi-Head Attention (MHA), Feed-Forward Network (FFN) and Normalization Layers (Norm). For each block, the input is a sequence of n vectors (tokens), denoted as $X \in \mathbb{R}^{n \times d_{in}}$, where n and d_{in} are the sequence length and the input feature dimension. Three linear projection weights $\{W_Q, W_K, W_V\} \in \mathbb{R}^{d_{in} \times d}$ project the input tensor X to Query, Key and Value tensors, denoted as $\{Q, K, V\} \in \mathbb{R}^{n \times d}$ (Step ① in Figure 2), where d represents the hidden feature dimension. The Q, K, V tensors are split to multiple "Heads" (Step ②) to perform softmax-based self-attention respectively (Step ③). The self-attention results are then concatenated (Step ④) and linear-transformed (Step ⑤) to generate the MHA results. After skip-connection and layer-normalization, the hidden feature X is then fed into the FFN layer, which computes with two fully-connected layers (Step ⑥ and ⑦). The GELU activation function [22] is applied to the first layer's output. One block's output serves as the input of the next block. On top of the last block, there is usually a classification layer to generate the final results for a given downstream task.

Traditional neural-networks like CNNs and LSTMs all involve "prior" in their models to enhance the performance (i.e., a CNN model assumes the 2D images have spatial locality, while an LSTM model assumes that the information should be either remembered or forgotten). In comparison, Transformers have no such priors and learn all the useful information

Table 1: Comparisons of State-of-the-Art Parameter-Efficient Transformers.

Model Type	Adapters [23]	MaskBert [64]	Diff-Pruning [20]	Bitfit [62]
Main Computation. Pre-trained Parameters Are Marked Blue				
Formula	$Y_t = X_t \cdot W + b,$ $Y_t = Y_t + \sigma(Y_t \cdot W_{down}) \cdot W_{up}$	$Y_t = X_t \cdot (W \odot M_t) + b$	$Y_t = X_t \cdot (W + \delta_t) + (b + b_t)$	$Y_t = X_t \cdot W + b_t$
Additional Parameters	7.3%	3%	0.5%	3.8%
Leading Tasks in GLUE Benchmark *	STS-B, QQP	SST-2	QNLI, MNLI, CoLA, MRPC	RTE

* Comparisons are based on BERT-large. We reproduce MaskBert on BERT-large, while the other results are obtained from the reported numbers.

purely through unsupervised pre-training. To achieve this, not only the pre-train datasets are large in scale, the models also contain enormous parameters to guarantee a high knowledge capacity. For instance, Bert-base and Bert-large have 110M and 340M parameters, respectively, while some recent models even have billions [3, 51] or trillions of parameters [13]. The explosion of such large-scale Transformer models brings both opportunities and challenges. On the one hand, real-world tasks get benefited from their superior performance compared to traditional DNNs. On the other hand, it is challenging to deploy Transformer models to resource-constrained scenarios due to the storage/memory capacity limit, especially when multiple tasks should be served simultaneously.

2.2 Multi-Task Transformers Serving

In real-world scenarios, a server usually runs multiple tasks (here a *task* refers to a distinct DNN model) concurrently for serving different queries [2, 48, 50] (each query invokes at least one of the DNN models). According to the standard workflow shown by steps ①-②-④ in Figure 1, for multi-task transformers serving, every downstream task has its own fine-tuned model. That is to say, the storage/memory overhead is proportional to the number of tasks. In the figure, three tasks occupy $3 \times$ storage. More importantly, all the models should be buffered in GPU memory for quick response to different queries. As the number of tasks increases, it will easily exceed the GPU's memory. Alternatively, we can swap in and out models once some tasks are invoked. Such a method, however, will downgrade the system's performance due to the considerable model swapping overhead [19, 48, 53]. Also, if each task only has limited input queries, the computation resources will be under-utilized because of the small batch size.

Although previous serving systems/frameworks like IN-FaaS [48], Nexus [50], Rafiqi [53], Triton [40], Tensorflow Serving [18] and many DNN accelerators [2, 6, 17, 26, 27] have emphasized the multi-task DNN serving ability, to implement multi-task Transformers serving is still challenging. Reasons are mainly two-folded. First, Transformers usually contain enormous parameters to guarantee their sufficient knowledge capacity. Thus, the storage and memory overhead is much heavier than traditional DNNs, limiting the number of served tasks. Second, previous multi-task inference frame-

works/accelerators assume that the computation/bandwidth requirements vary among concurrent DNNs. Thus, they execute computation-bounded and memory-bounded models (or layers) together to fully utilize the hardware resources. However, since the Transformer blocks are homogeneous among different tasks, there is little room for improving system throughput by co-locating heterogeneous models.

2.3 Parameter-Efficient Transformers

A potential solution to the multi-task Transformers serving problem is directly training a multi-task model like T5 [46]. However, such a method is infeasible in real scenarios since all the application developers have to provide their private datasets to train such a one-for-all model. Recently, Parameter-Efficient Transformers (PETs) have emerged as another promising way to deal with the problem. PETs are based on the assumption that pre-trained models have learned rich knowledge from large-scale pre-train datasets [44, 47]. Thus, we can adapt the pre-trained model to downstream tasks by only fine-tuning a small portion of task-specific parameters rather than the whole model. As illustrated in Figure 1, through parameter-efficient fine-tuning (Step ②), only the PET parameters should be stored for each downstream task. For example, four representative PETs, namely Adapters [23], MaskBert [64], Diff-Pruning [20], and Bitfit [62] only use 0.5% to 7.3% additional parameters for each task. However, they still achieve comparable or even higher accuracy against the full-model fine-tuning counterparts. We summarize them in Table 1 and introduce them as follows:

Adapters: Adapters [23] proposes to inject trainable, task-specific "adapter" modules between some layers of the pre-trained model, while the pre-trained weights are shared among tasks. Formally, assume the linear layers in a pre-trained model compute the hidden feature Y_t with input feature X_t and pre-trained parameters W (weight) and b (bias), namely $Y_t = X_t \cdot W + b$, then an adapter module manipulates the hidden features with two learnable weights $W_{down} \in \mathbb{R}^{d \times d_m}$ and $W_{up} \in \mathbb{R}^{d_m \times d}$, namely $Y_t = Y_t + \sigma(Y_t \cdot W_{down}) \cdot W_{up}$, where σ is the activation function. Since the bottleneck dimension $d_m \ll d$, the Adapter modules are small in size. Each task only requires about 7.3% of new parameters (including a task-specific classification layer).

MaskBert: Based on the lottery ticket hypothesis on Bert [5, 43], MaskBert [64] adapts the pre-trained model to downstream tasks by learning binary masks for each weight matrix. As shown in Table 1, for each task, the pre-trained model (including the classification layer) is frozen. Only the binary masks with about 5% of zero elements are learned for each weight matrix. Since the masks are binary, MaskBert only incurs about a 3% per-task storage overhead. For each linear layer, the computation is represented as $Y_t = X_t \cdot (M_t \odot W) + b$, where M_t denotes the task-specific mask.

Diff-Pruning: Diff-Pruning [20] also shares the pre-trained model among tasks and only fine-tunes a small portion of "difference" for each downstream task. As shown in Table 1, the orange elements in both weight and bias represent the fine-tuned "difference", which only incur about 0.5% of new parameters for each task. During inference, these difference parameters, denoted as δ_t and b_t , are merged with the pre-trained model to construct a task-specific model. Thus, the main computation is $Y_t = X_t \cdot (W + \delta_t) + (b + b_t)$.

Bitfit: Besides a task-specific classification layer, Bitfit [62] only fine-tunes the linear and normalization layers' bias-terms, which also achieves competitive accuracy on some tasks in the standard GLUE benchmark [54]. As shown in Table 1, the linear layers in Bitfit compute with $Y_t = X_t \cdot W + b_t$ where b_t is the only task-specific parameter.

There are still many other emerging PET algorithms [14, 24, 33, 41]. Their workflows are similar to at least one of the above PETs. Therefore, in this paper, we conduct the discussion mainly based on these four representative PETs.

2.4 Challenges of Multi-Task PETs Serving

When serving T different tasks, PETs reduce the storage overhead from original $T \times \gamma$ to $T \times \eta + \gamma$, where γ and η denote the amount of full-model parameters and PET parameters, respectively. Since $\eta \ll \gamma$, using PETs can significantly reduce the storage overhead. However, we notice that the algorithmic advantages of PETs can hardly translate to real speedup with conventional Transformers serving frameworks, mainly due to the following challenges:

Challenge #1: Current frameworks cannot support various PET algorithms flexibly. We present the leading GLUE tasks of each PET algorithm in Table 1. As we can see, all PETs have their advantageous tasks. None of the four PETs can serve as the one-for-all choice. That is to say, the application developers tend to choose the best PETs for their downstream tasks [33]. Therefore, the serving framework has to support multiple types of PETs. However, current serving frameworks are not optimized for diverse PETs. They lack the mechanism to register and manage different PETs flexibly, considering their distinct algorithmic representations.

Challenge #2: The GPU-memory footprint is still not mitigated. To serve PETs like MaskBert and Diff-Pruning using conventional inference frameworks, we have to merge the task-specific PET parameters into the shared model. Af-

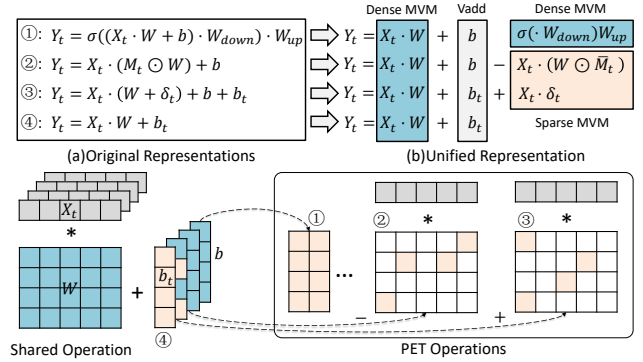


Figure 3: Unified representation of PETs. ①: Adapters ②: MaskBert ③: Diff-pruning ④: Bitfit.

ter that, we load the newly constructed models to inference frameworks for serving. Thus, concurrent tasks still occupy $O(T \times \gamma)$ GPU memory, limiting the system's scalability. As discussed before, we can swap in/out models among tasks to deal with the GPU capacity issue, which will result in low throughput due to the model swapping overhead [19, 48, 53].

Challenge #3: It is still hard to improve the system's serving throughput, especially when each task only has limited queries. It is well-known that batched inference is a practical technique to improve a DNN serving system's throughput [7, 12, 16, 50]. However, due to the differences in PET parameters and PET algorithms, conventional frameworks can hardly batch different tasks' queries (even though the tasks may belong to the same PET algorithm) for higher throughput. Such a problem will be more prominent when each concurrent task only has a few queries to process.

3 PetS Framework

To address the challenges outlined above, we propose PetS, a unified framework for efficient multi-task PETs serving. We first propose a unified representation to put all PETs into one framework. Based on this, we develop a flexible PET tasks management mechanism and a specialized PET Inference Engine (PIE) that enables both inter-task and inter-algorithm query-batching. Details are introduced as follows.

3.1 Unified Representation of PETs

As Table 1 shows, state-of-the-art PETs have different algorithmic representations, resulting in a "fragmentation" problem. We propose a unified representation, which expresses PETs with task-agnostic and task-specific operations, to help put them into one framework and enable batched inference. As illustrated in Figure 3, for each PET, we decouple the main computation (linear layers) into three operations: (1) Dense Matrix-Vector-Multiplication (MVM) operation using shared pre-trained weights. (2) Bias vector addition (Vadd) using shared or task-specific bias. (3) Sparse/dense MVM operations using task-specific PET parameters. Since all PETs share the same pre-trained weight matrix W , the first operation, namely $X_t \cdot W$ can be batched together. Though the task-

specific computation with PET parameters cannot be batched among PETs, it only involves light-weighted operations.

Adapters and Bitfit naturally fit into such a representation, since the PET operations are already decoupled from the shareable operations. For Diff-Pruning and MaskBert, we need to perform some equivalent transformations. As Figure 3 shows, for Diff-Pruning, the computation concerning the shared weight and "difference" are conducted separately. Then the results are added up, namely $X_t \cdot (W + \delta_t) = X_t \cdot W + X_t \cdot \delta_t$. For MaskBert, we use an equivalent transformation: $M_t \odot W = (1 - \bar{M}_t) \odot W$, where \bar{M}_t denotes the bit-wise inversion of binary mask M_t . Thus, the original MVM operation is converted to $X_t \cdot W - X_t \cdot (W \odot \bar{M}_t)$. The $W \odot \bar{M}_t$ term can be treated as the sparse weight differences similar to Diff-Pruning's. Since δ_t and $W \odot \bar{M}_t$ are sparse matrices with high sparsity (typically 95% - 99.5%), these PET operations can be efficiently computed with sparse kernels. Considering that the Vadd operations concerning bias terms only have little overhead, we mainly focus on operations (1) and (3), namely the sparse and dense MVM operations.

The unified representation brings two main advantages. First, the queries from different tasks can be batched together at step (1), regardless of the PET types. Let us consider an extreme case: assume we have T tasks, each having a single query. The execution latency is changed from $\sum_{i=0}^{T-1} \alpha(1)$ to $\sum_{i=0}^{T-1} \beta_i + \alpha(T)$, where $\alpha(n)$ denotes the latency of running a Transformer model (without PET) with a batch of n queries. β_i denotes the latency of PET operations for query i . The throughput is improved if we have:

$$\sum_{i=0}^{T-1} \beta_i + \alpha(T) < \sum_{i=0}^{T-1} \alpha(1) \quad (1)$$

The inequality always holds since $\beta_i \ll \alpha(1)$ (the PET operations are light-weighted compared to the shared operations) and $\alpha(T) \ll \sum_{i=0}^{T-1} \alpha(1)$ (batched inference can greatly reduce the average latency [7, 12, 16, 50]).

Second, such a unified representation simplifies the PET tasks management. Each task can be registered by identifying its shared model tag, PET type, and PET parameters. The inference engine can then load these PETs in a unified way.

3.2 Framework Overview

Based on the unified representation, we then present the PetS serving framework to support the management and serving of PET tasks. Figure 4 illustrates the proposed PetS framework. PetS has three main components: a Task Manager, a Parameter Repository, and a PET Inference Pipeline. PetS works as follows: ❶: The framework first registers the PET tasks submitted by developers. For each PET task, the developers are required to provide the Pre-trained Model Tag (such as bert-base-cased), PET Parameters (in compressed format) and PET Type (e.g., MaskBert). ❷: Task Manager registers PET tasks, which assigns a unique Task_id to each submitted task. The PET parameters and the

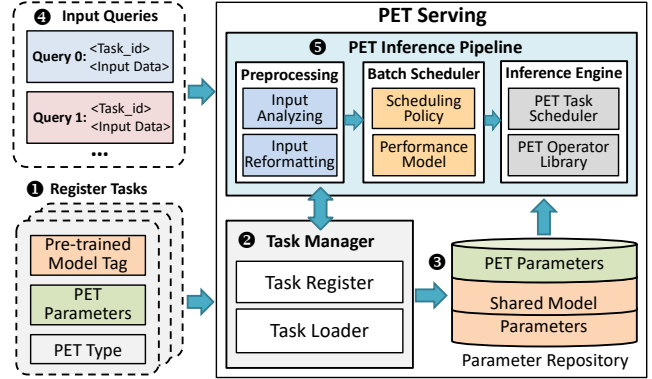


Figure 4: PetS System Overview.

pre-loaded shared model parameters are all stored in the Parameter Repository (❸). After registration, the PET Inference Engine (PIE) is responsible for processing the input queries (❹) through an optimized PET Inference Pipeline (❺).

3.3 Managing PET Tasks

One of the key features of PetS is the flexible and efficient PET task management mechanism, which is powered by the Task Register and Task Loader modules.

Task Register: The Task Register module registers a PET task according to the user-provided information denoting its shared model, task-specific parameters, and PET type. A triplet $\langle \text{Task_id}, \text{Shared_model_tag}, \text{PET_type} \rangle$ is formed to bind each task with its corresponding pre-trained model and the supported PET type, where the Task_id is unique to identify each PET task. All these triplets are organized as a map structure, with the Task_id as the key and the $\langle \text{Shared_model_tag}, \text{PET_type} \rangle$ pair as the value. Therefore, we can index the metadata of each task given the Task_id of a query. Once a PET task completes registration, its PET parameters are stored in the Parameter Repository. Note that for PETs like MaskBert and Diff-Pruning, the PET parameters are stored in a compressed format to save storage. **Task Loader:** Before a PET task is invoked by the inference engine, the Task Loader module firstly loads the shared model parameters if they have not been loaded yet. Otherwise, the Task Loader indexes the Parameter Repository and accesses the PET parameters according to the Task_id of each invoked task. Considering that the PET parameters are small in size and the shared model only has one copy, all the parameters can be buffered into the GPU memory for quick invoking.

3.4 PET Inference Pipeline

At the core of PetS framework is the PET Inference Pipeline, which processes queries with three pipelined steps including Preprocessing, Batch Scheduling and PET Inference.

3.4.1 Preprocessing

The preprocessing module fetches queries from standard HTTP/gRPC data plane similar to conventional inference serving frameworks [18, 40]. Then it analyzes input data

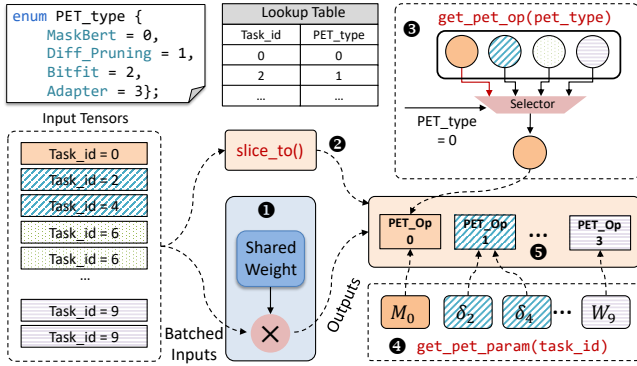


Figure 5: Base PET inference engine workflow.

and reformats them for the next query-batching step. Firstly, the input data is classified according to the shared model (Shared_model_tag). The metadata of each query is extracted, such as the invoked task’s id, sequence length, PET type, etc. Some preliminary data preprocessing operations are then performed according to the extracted metadata, such as grouping the queries of the same PET task. In order to improve the performance of sparse PET operations, the input tensors’ data layout is also reformatted according to the compression format of the corresponding sparse PET parameters. Finally, the preprocessed input queries together with the extracted metadata are dispatched to different queues for further scheduling, according to their targeting shared models.

3.4.2 Batch Scheduler

As discussed before, batching is an effective way to improve system throughput. Though `PetS` enables both inter-task and inter-algorithm batching through the proposed unified representation, the heterogeneity of queries in terms of PET type and sequence length still prevents batching efficiently. The batch scheduler module is used to overcome the challenge posed by query heterogeneity. Taking preprocessed queries as input, the batch scheduler tries to maximize the benefit of batching PET operators and minimize the padding overhead of batching shared operators with different sequence length at the same time. It leverages an accurate performance model to help make batching decisions. The details of the scheduling policy will be described in Section 4.1.

3.4.3 PET Inference Engine

PIE Workflow: The batched queries are finally fed into the PET Inference Engine (PIE). Figure 5 illustrates the base workflow of PIE. In the Figure, we use different colors to indicate the queries’ PET types in the batch. For example, task 0 belongs to MaskBert, while tasks 2,4 belong to Diff-Pruning. PIE starts the computation of each Transformer layer as follows: ❶: PIE performs batched GEMM computation using the input tensor and shared weights W . ❷: PIE gets the `PET_type` attributes of each query by searching in the lookup table with its `Task_id`. The batched inputs are also sliced into several mini-batches (intra-task batching) according to the task id. Then PIE gets the PET operators (❸) and PET

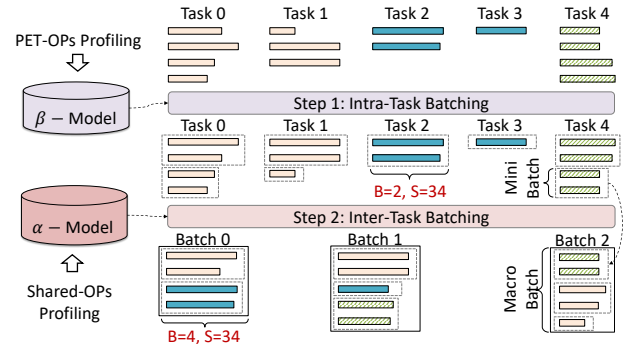


Figure 6: Coordinated Batching Strategy

parameters (❹) according to the obtained PET types. ❺: PIE executes the PET operators successively on each sliced mini-batch. These PET operators are also responsible for adding the PET results to the shared outputs if needed. Note that the remaining operations like self-attention are also performed by PIE but are not shown in the figure.

PET Operator Library: According to the unified representation in Figure 3, the PET tasks rely on different PET operations. MaskBert and Diff-Pruning involve sparse matrix-matrix multiplication (SpMM) as their PET parameters contain high sparsity. Adapters perform light-weighted dense GEMM. While Bitfit only requires a Vadd operation. Therefore, PIE provides an operator library containing high-performance implementations of both dense and sparse operators. The sparse operators are tuned specifically for the sparse patterns and parameter sizes of the target models. We can also implement new PET operators for other emerging PET algorithms if they can fit into the unified representation.

PET Task Scheduler: During the inference of each layer, the PET operations of different tasks have no data dependency and can therefore run in parallel. Given the system allowed parallelism, e.g., the number of CUDA streams on GPU, the PET task scheduler schedules the PET operations to utilize the parallelism as much as possible. The PET operator scheduling strategy is introduced in Section 4.2.

4 Optimization Strategies

4.1 Coordinated Batching

In real scenarios, the input queries usually have variable sequence lengths. If we batch short queries and long queries, the short ones have to be zero-padded, incurring useless computation. Previous frameworks like TurboTransformers [12] pay much attention to solve such a problem. However, for `PetS`, we have to consider both the shared operations and PET operations. Therefore, we propose a Coordinated Batching (CB) strategy to coordinate these two parts during batching.

Problem Formulation: Assume there are R queries, namely $Q = \{x_0, x_1, \dots, x_{R-1}\}$ associated with T different tasks. We divide queries into M batches. For each batch, we use $\alpha[N][L]$ to denote the shared model latency when batching N queries with a maximum length of L . In the meantime, a PET operator

Algorithm 1: Coordinated Batching Strategy

```
1 Input: Number of tasks  $T$ , queries  $Q = \{x_0, x_1, \dots, x_{R-1}\}$ , Shared  
   Op latency model  $\alpha$ , PET Op latency model  $\beta$ ;  
2 Step 0: Pre-processing  
3 Cluster input queries to the same task and generate  
    $Q = \{X_0, X_1, \dots, X_{T-1}\}$ , where  $X_i$  contains  $n_i$  queries;  
4 Step 1: Intra-task batching  
5 for  $i \leftarrow 0$  until  $T$  do  
6   Create DP state vector  $state[n_i + 1]$ ,  $state[0] = 0$ ;  
7   Sort queries in  $X_i$  according to the sequence length in an  
   ascending order;  
8   Create  $split\_idx\_list[n_i + 1]$ ,  $pt = get\_pet\_type(i)$ ;  
9   for  $j \leftarrow 1$  to  $n_i$ ;  $min\_cost = INF$  do  
10    for  $k \leftarrow 1$  to  $j$  do  
11      $tmp = state[k - 1] + \beta[pt][j - k + 1][X_i[j].len]$ ;  
12     if  $tmp < min\_cost$  then  
13       $min\_cost = tmp$ ,  $split\_idx = k - 1$ ;  
14      $state[j] = min\_cost$ ,  $split\_idx\_list[j] = split\_idx$ ;  
15   Split queries into mini-batches  $MB$  using  $split\_idx\_list$ ;  
16 Step 2: Inter-task batching  
17 Sort mini-batches according to their max sequence lengths;  
18 Create DP state vector  $state[\#mini\_batch + 1]$ ,  $state[0] = 0$ ;  
19 Create  $sum[\#mini\_batch + 1]$ ,  $sum[i]$  records the total queries of the  
   first  $i$  mini-batches;  
20 for  $i \leftarrow 1$  to  $\#mini\_batch$ ;  $min\_cost = INF$  do  
21   for  $j \leftarrow 1$  to  $i$  do  
22     $batch\_size = sum[i] - sum[j - 1]$ ;  
23     $tmp = state[j - 1] + \alpha[batch\_size][MB[i].max\_seq\_len]$ ;  
24    if  $tmp < min\_cost$  then  
25      $min\_cost = tmp$ ,  $split\_idx = j - 1$ ;  
26    $state[i] = min\_cost$ ,  $split\_idx\_list[i] = split\_idx$ ;  
27 Split mini-batches into macro batches using  $split\_idx\_list$ ;  
28 Return: The scheduled macro batches
```

takes $\beta[pt][n][l]$ seconds to process the PET terms of n queries, whose PET_type is shortened to pt , and l is the max length of these n queries. Then, the estimated execution latency is:

$$Batch_Latency(B_i) = \alpha[N_i][L_i] + \sum_{j=0}^{t_i-1} \beta[pt_{ij}][n_{ij}][l_{ij}]. \quad (2)$$

In the formula, we use B_i to denote the i -th batch, and assume there are t_i different tasks in the batch. For the j -th task in batch i , there are n_{ij} queries that shape a mini-batch, which is processed by a PET operator indexed by pt_{ij} . The longest sequence length of the n_{ij} queries is l_{ij} . Since all the M batches, namely $\mathcal{B} = \{B_0, B_1, \dots, B_{M-1}\}$ are executed successively, we can further estimate the total latency as:

$$\begin{aligned} Total_Latency(\mathcal{B}) &= \sum_{i=0}^{M-1} Batch_Latency(B_i) \\ &= \sum_{i=0}^{M-1} \alpha[N_i][L_i] + \sum_{i=0}^{M-1} \sum_{j=0}^{t_i-1} \beta[pt_{ij}][n_{ij}][l_{ij}] \end{aligned} \quad (3)$$

As we can see, the total latency is jointly determined by the shared and PET operators. To coordinate these two parts, we propose a two-step Coordinated Batching strategy. As illustrated in Figure 6, in the first step, we generate "mini-batches" for each task (intra-task batching), which only considers the effect of batching PET operators using a profiled β -model. In the second step, we generate "macro-batches" by combining these mini-batches among tasks (inter-task batching), which

Algorithm 2: PET Operator Scheduling Strategy

```
1 Input: PET operator set  $\mathcal{O}$  of a macro-batch, stream set  $\mathcal{S}$ ,  
   latency model  $\beta$  and bandwidth model  $\omega$   
2 Output:  $\mathcal{O}$  to streams assignment  $\Phi(\mathcal{O} \rightarrow \mathcal{S})$   
3  $I \leftarrow \emptyset$   $\triangleright$  operational intensity of  $\mathcal{O}$ ;  
4 for  $o \in \mathcal{O}$  do  
5    $op\_intensity \leftarrow (\frac{o.FLOPs}{\beta(o)}) / \omega(o)$ ;  
6    $I.append(op\_intensity)$ ;  
7 Sort  $\mathcal{O}$  in an ascending order according to  $I$ ;  
8 for  $o \in \mathcal{O}$  do  
9    $stream\_idx \leftarrow \lfloor o.idx / |\mathcal{S}| \rfloor$ ;  
10   $\Phi(\mathcal{O}[o.idx]) \leftarrow \mathcal{S}[stream\_idx]$ ;
```

only considers the effect of batching shared operators using a profiled α -model. In both steps, we sort the queries and use dynamic programming (DP) to find the optimal splitting positions with low time-complexity.

Algorithm 1 details this strategy. The queries with the same task id are firstly clustered and sorted according to the sequence length. At the first step, we use $state[i]$ to record the minimum latency of PET operations when batching the first i queries. We use $split_idx_list$ to record the splitting positions. Equation 4 shows the Bellman equation:

$$state[i] = \min_{0 < j \leq i} (state[j - 1] + \beta[pt_{ij}][i - j + 1][l_{ij}]) \quad (4)$$

With the DP algorithm, we divide the queries of each task into mini-batches. At the second stage, the Bellman equation only considers the shared operators, whose latency is estimated by the α model. Instead of scheduling each single query, the second step schedules the mini-batches:

$$state[i] = \min_{0 < j \leq i} (state[j - 1] + \alpha[batch_size][L_j]) \quad (5)$$

Where $state[i]$ records the minimum latency of batching the first i mini-batches. L_j denotes the max sequence length of the j -th mini-batch. $batch_size$ denotes the number of total queries from mini-batches i to j . After dynamic programming, the mini-batches are assigned to multiple macro-batches.

4.2 PET Operator Scheduling

In addition to the coordinated batch scheduling, PET operators can be executed in parallel to further improve hardware utilization and performance. To achieve the PET-task-level parallelism on GPU, PET operators in a macro-batch (as shown in Figure 6) can be assigned to multiple CUDA streams. However, naively assigning a unique stream to each PET task may not get the ideal speedup, because if we assign computation-intensive operators to different streams (or memory-intensive operators), they can hardly be executed in parallel, since they are bounded by the same resources. Therefore, we propose a light-weighted online scheduling strategy to dynamically assign PET tasks to streams. The scheduling algorithm is shown in Algorithm 2. The input includes the set of PET operators to be scheduled, and the set of streams on which the PET operators execute. The algorithm also requires the PET latency model β used in Algorithm 1, as well as a bandwidth

Table 2: PET data structures and interfaces

Data Structure	Interface
PETModel	load_shared_model(model_url) load_pet_task(pet_type, param_url)
PETLayer	load_pet_params(pet_type, pet_layer_param)

model ω generated together with β . The algorithm outputs the assignment from the PET operator set to the stream set. The first step (lines 3–6) of Algorithm 2 computes the operational intensities of all the PET operators. Operational intensity is a metric to measure the compute to memory access ratio of an operator [57]. The achieved intensity of an operator is computed by its FLOPs divided by the utilized bandwidth. The second step (lines 7–10) then assigns a stream for each PET operator. The rationale is to put operators with differentiated operational intensities to different streams, in order to minimize resource conflict between streams.

Though the PET operator level parallelism contradicts the assumption that the PET operators are executed sequentially in Coordinated Batching. Experiments in Section 6 demonstrates that the coordinated batching still works well with parallel PET execution. Involving a more accurate performance model for parallel PET execution can help generate better scheduling results. We leave this as our future work.

5 Implementation

We implement `PetS` with a Python front-end to describe shared model and PET tasks management, and a C++ backend to perform query scheduling and inference serving.

5.1 PET Description

The description of PET tasks is based on the HuggingFace Transformers framework [58]. We extend HuggingFace Transformers library mainly with two data structures and three interfaces to manage PET tasks, as shown in Table 2. `PETModel` is the base structure to implement a model with PET tasks. `PETLayer` is defined in `PETModel` to describe PET operations, apart from the shared operations. The first interface of `PETModel` in Table 2, `load_shared_model`, loads the shared parameters as traditional HuggingFace tasks do. The `load_pet_task` interface is used to load PET parameters, given PET type and PET parameters URL. It will call `load_pet_params` defined in `PETLayer` to finish the underlying load operations for each layer. Users can inherit and implement these interfaces according to specific tasks.

5.2 Inference Serving

The three modules of the `PetS`'s PET Inference Pipeline in Figure 4 are deployed in individual processes to process input queries in a pipelined manner.

Inference Engine: Inference frameworks compatible with HuggingFace Transformers library can be plugged into `PetS` as its backend engine, such as TurboTransformers [12], LightSeq [56], FastTransformers [37], etc. Modifications should be done to support PET operators and the PET Task

Code Listing 1: User Interface

```
server = PetS() # create a PET server
# Register PET tasks
server.register_task("Adapter", "bert-base", pet_param_url_0)
server.register_task("MaskBert", "bert-base", pet_param_url_1)
# Register other PET tasks ...
# Load shared model parameters and PET tasks
server.load_shared_model("bert-base")
server.load_pet_tasks(pet_task_ids)
# Fetch queries from input query queue and run inference.
queries = server.fetch(input_query_queue)
results = server.inference(queries)
```

Table 3: Shared Model Configurations

Bert Type	#Layer	#Head	Hidden Size	Inter-Size	# Params
DistillBert	6	12	768	3072	66 M
Bert-base	12	12	768	3072	110 M
Bert-large	24	16	1024	4096	340 M

Scheduler. Without loss of generality, `PetS` implements the backend inference engine based on TurboTransformers*. It leverages cuBLAS to compute the shared dense MVM operators. We leverage a high-performance SpMM implementation [15] to implement the sparse PET operators.

5.3 User Interfaces

We use a code sample as shown in Code 1 to demonstrate how users can launch `PetS`, load models and process queries with only a few lines of Python code.

After creating a `PetS` server, it firstly registers PET tasks through the `register_task` interface. It will write the user-provided PET parameters to Parameter Repository and get the assigned task ids. Then the server loads the shared models by calling standard HuggingFace Transformers API and loads PET tasks using the assigned task ids to index the Parameter Repository. Once fetching a group of queries from the input query queue, the `PetS` server runs the PET inference pipeline and returns the inference results.

Currently, we only implement the four aforementioned PET algorithms in the `PetS` framework. A new PET algorithm can work with `PetS` as long as it meets two requirements: (1) The PET operations are separable (with necessary equivalent transformations) from the shared operations. (2) The separated PET operations are light-weighted. Then, to support a new algorithm, the developers should first identify its PET operations. Then the related functions introduced in this section should be extended accordingly.

6 Evaluation

6.1 Experimental Setup

Shared Models: We choose Bert-base, Bert-large [9], and DistillBert [49] as the shared pre-trained models, whose configurations are listed in Table 3. We do not include generative Transformer models such as GPT-2 [45] because GPT-like models have not been well-studied by the PET algorithms discussed above. We leave the evaluation on GPT-like models as our future work and focus on Bert-like models in this

*<https://github.com/Tencent/TurboTransformers>

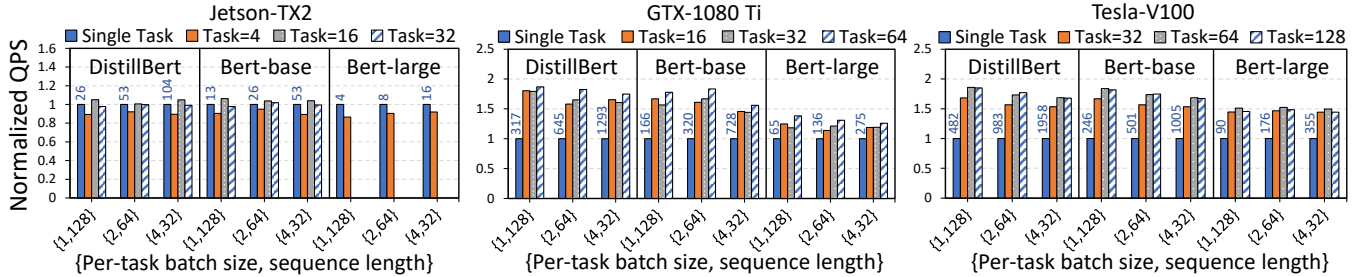


Figure 7: Throughput Improvement Evaluation on Multiple Platforms.

Table 4: PET Configurations

PET Type	Configuration	Main PET Params
Adapter	Bottleneck = 64	W_{down} and W_{up} at the BertOutput, SelfAttenOutput layers.
MaskBert	95% Sparsity*	Binary masks for all the linear weights.
Diff-Pruning	99.5% Sparsity	Sparse difference concerning linear weights, bias terms, and the classification layer.
Bitfit	N/A	Bias terms of linear / layernorm layers and a classification layer.

* Obtained through equivalent transformation

Table 5: Profiling of Total Supported Tasks

Platform	Device Memory	Shared Models	DistillBert	Bert-base	Bert-large
			SeqS / PetS	SeqS / PetS	SeqS / PetS
Jetson TX2	8GB*	Supported Tasks	34 / 504	17 / 180	3 / 12
GTX-1080Ti	11GB		56 / 1336	28 / 588	7 / 126
Tesla-V100	32GB		170 / 4344	85 / 2164	25 / 560

* Shared by CPU and GPU

paper. Note that although in Section 2 we mainly use a single layer for illustration, the evaluations in this section are all conducted on entire models.

PET Tasks: The configurations of four PET algorithms are summarized in Table 4. For Adapter, we set the hidden size (d_m) of the adapter modules to 64. For MaskBert and Diff-Pruning the PET parameters’ sparsity is set to 95% and 99.5%, respectively (we use the equivalent transformation proposed in Section 3.1 to obtain the 95% sparsity for MaskBert). For all PETs, we reproduce the algorithm on HuggingFace Transformers to obtain the trained PET parameters.

Platforms: We evaluate PetS on Edge/Desktop/Server platforms, namely Jetson TX2 (8GB memory, shared by CPU and GPU), GTX-1080Ti-11GB (Intel Xeon E5-2690 CPU), and Tesla-V100-32GB (Intel Xeon Golden 5220 CPU, two sockets). The V100 platform installs CUDA-10.1. The 1080 Ti platform installs CUDA-11.3. The TX2 platform is flashed with Jetpack 4.4.1 containing CUDA-10.2.

6.2 Main Results

6.2.1 Maximum Number of Supported Tasks

We first demonstrate PetS’s scalability by comparing the maximum number of supported tasks with conventional Sequential Serving Systems (SeqS) [1, 7, 12]. Without loss of generality, here we use the unmodified TurboTransformers framework as a representative for SeqS. SeqS loads full-model copy for each task, while PetS works on light-weighted PET tasks. For each platform, we load T tasks (for PetS, each task belongs to a random PET type). If the system can process a batch of

32 randomly-generated queries (each query has a length of 128) without the out-of-memory (OOM) issue, we assume the system can support at least T tasks. We increase T repeatedly to test the limit. The maximum supported tasks are listed in Table 5. Compared to conventional SeqS systems, PetS supports $4\times$ (Bert-large on TX2) to $26\times$ (DistillBert on V100) more concurrent tasks, thanks to the proposed unified representation and efficient PET tasks managing mechanism. Therefore, PetS can substantially save the hardware cost when deploying multiple Transformer-based applications to scenarios from edge computing to cloud computing. Also, it avoids the notoriously slow model swapping [19, 48, 53] even when hundreds to thousands of tasks are invoked.

6.2.2 Throughput Improvement

As stated before, PetS achieves both inter-task and inter-algorithm batching through the unified representation and a specialized PET Inference Engine (PIE). Therefore, we evaluate PetS’s throughput (measured in Queries-Per-Second, QPS) under different situations. As shown in Figure 7, we load 4~32, 16~64 and 32~128 random tasks on TX2, 1080 Ti and V100 platforms, respectively. For each task, we generate queries with three fixed shapes. All queries with the same shape are executed in one batch. We adopt SeqS running a single task as the baseline. Note that here we do not include the two optimizations introduced in Section 4 and adopt a simple fixed-batch policy in the batch scheduling step. We assume that there are no dependencies between tasks.

As we can see, on the 1080 Ti and V100 platforms, PetS achieves up to $1.87\times$ and $1.86\times$ higher throughput, $1.53\times$ and $1.63\times$ on average, compared to the single-task serving baseline. We notice that PetS fail to achieve meaningful speedup than single-task serving on TX2. This is because TX2 only has limited computation resources (256 CUDA cores) and therefore can hardly get benefited from batched inference. Similarly, on 1080 Ti and V100, we observe lower speedup on Bert-large models than Bert-base/DistillBert. This is because Bert-large has a larger layer size (see Table 3), which saturates the GPUs’ computation resources more easily, diminishing the benefits of batched inference.

6.2.3 Comparison with ParS

Apart from Sequential Serving Systems (SeqS), several previous serving systems are built to support concurrent execution of multiple tasks in parallel [18, 40, 48], which belong to the

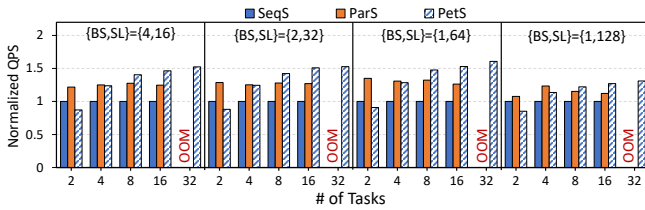


Figure 8: Throughput Comparison with SeqS and PARS.

Parallel Serving System (PARS). We compare PetS’s throughput over conventional PARS to examine the performance. To implement PARS, we modify the original TurboTransformers framework and put each task-specific model to a unique CUDA stream to run all models in parallel. Note that in this experiment, we adopt Bert-base as the shared model and do not consider model swapping. All results are collected on the GTX-1080 Ti platform.

As shown in Figure 8, we evaluate multiple query configurations represented by a pair of per-task batch size and sequence length ($\{BS, SL\}$) to illustrate the PetS’s generality. We run each query configuration on a different number of tasks for SeqS, ParS, and PetS. All results are normalized to the SeqS baseline. When the number of tasks is small (1~4), PetS cannot outperform ParS. For one thing, PetS has extra PET operations, as illustrated in Figure 3. For the other, tasks of ParS run in parallel. Although the shared weight part of PetS can also utilize parallel hardware, the overhead of PET operations cannot be offset by the limited parallelism. As the number of tasks increases, the benefit of PetS begins to manifest. PetS has an average 17.7% speedup over ParS when the number of tasks reaches 16 on all four configurations. Neither SeqS nor ParS could run too many concurrent tasks due to OOM, while PetS is still able to scale to 32 tasks and even more (refer to Table 5). As we can find in the figure, benefited from the higher hardware utilization, the QPS of PetS improves with the increased total batch size (i.e., $\#tasks \times BS$). As the number of tasks further increases to 256 or more (not shown in the figure), the QPS improvement curve will reach a plateau since a large batch saturates the GPU resources.

6.3 Performance Analysis

6.3.1 Execution Time Breakdown

To figure out why PetS outperforms the baseline systems in serving throughput, we break down the execution time of both PetS and SeqS on GTX-1080 Ti. We set two workloads (i.e., per-task batch size = 1, sequence length = 64 and per-task batch = 2, sequence length = 32) and evaluate eight random tasks with Bert-base and Bert-large models. Therefore, the two workloads issue 8 and 16 queries each time. As we can see in Figure 9, PetS speeds up the Non-PET operators (including the attention operations and the computation of shared linear layers) by $2.17\times$ to $3.28\times$, thanks to the batched execution of shared operators. Due to the adoption of SpMM library, the PET operators only take up 27.4% to 41.3% of the total execution time. Therefore, the end-to-end execution

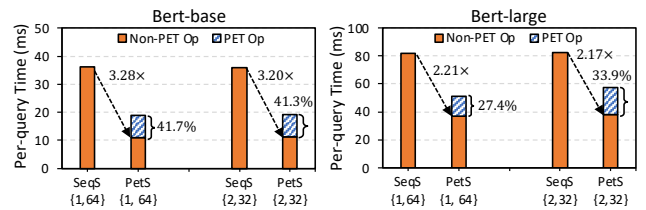


Figure 9: Execution Time Breakdown.

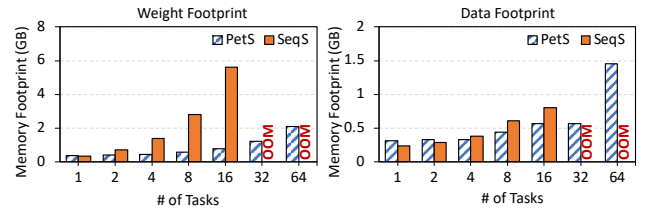


Figure 10: GPU Memory Footprint Comparison.

time is still much less than SeqS.

6.3.2 Memory Footprint Breakdown

We also profile the memory footprint of PetS and SeqS on 1080 Ti to understand why PetS has outstanding scalability. Taking the configuration of $\{BS, SL\} = \{1, 128\}$ as an example, we plot the consumed GPU memory by both model weights and data under different task numbers in Figure 10.

We can see that a single task has about 0.35GB weight parameters. The memory consumption of SeqS grows linearly with the number of tasks. The weight parameters exceed 11GB for 32 tasks on SeqS, causing OOM on GTX-1080 Ti. On the contrary, for PetS, only the memory footprint of the PET parameters, which normally occupies less than 5% of the shared weight, increases with the number of tasks. As a result, the total memory footprints of 64 tasks occupy less than 40% of total GPU memory, demonstrating that PetS can support much more tasks. Note that the memory footprints of 16 and 32 tasks are the same, but 64 tasks consume three times more data memory than 32 tasks. This is because we use NVIDIA CUB device memory allocator [36] for dynamic data memory management, and the allocated device memory is not strictly proportional to data size, but aligned according to some rounding rules.

6.3.3 Effect of PET Operator Scheduling

In Section 4.2, we introduce a PET Operator Scheduling strategy to properly schedule PET operators to multiple CUDA streams. To demonstrate the benefits of such a strategy, we also profile the speedup using Bert-base on GTX-1080 Ti GPU. We set the sequence length from 4 to 64. For each test case, we put the same 1024 random queries in the pool and process them with a batch of 128. As shown in Figure 11, when 32 tasks are served, increasing the number of streams to 32 brings the optimal performance for all input configurations and reduces up to 15% of execution latency. However, we observe that as the number of tasks keeps growing to 64 and 128, using more than two streams even downgrades the performance under the Seq = 4 configuration. The main inferred

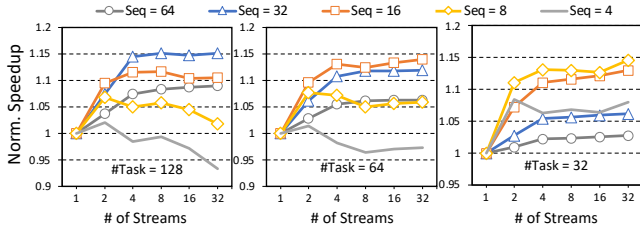


Figure 11: PET Operator Scheduling Performance.

reason is that, as the number of tasks increases, each task’s batch size gets smaller. When the input queries also have limited sequence lengths (i.e., seq = 4), the execution time is too short for the GPU’s scheduler to overlap concurrent streams. Thus, launching too many streams will only incur non-negligible synchronization overhead.

For PIE, the number of CUDA streams can be dynamically set before each batch’s inference. Therefore, we can add a rule to the PET operator scheduler: for tasks with short sequence lengths and small per-task batch size, we set a small stream number such as one or two. For tasks with a long sequence length, we set a large stream number such as 32. We obtain the threshold on each GPU platform by profiling in advance.

6.4 Performance on Arbitrary Inputs

As stated before, in real-world multi-task serving scenarios, the input queries usually have variable sequence lengths and PET types. Naïvely batching these queries may not bring the ideal throughput. Our proposed Coordinated Batching (CB) strategy is aimed to improve *PetS*’s performance on arbitrary inputs by coordinating shared operations and PET operations during batching. To evaluate the effect of CB, we test on workloads with variable sequence lengths and PET types, and then compare CB with three baseline batching strategies:

Fixed-Sized Batching: We put queries in the pool to fix-sized batches, regardless of their PET types and sequence lengths.

α -only Batching: We dynamically batch the queries only using the α model. This strategy is similar to TurboTransformers’ smart batching. To implement the α -only Batching, we treat every single query as a mini-batch and only conduct the inter-task batching (step 2) in Figure 6.

β -only Batching: We dynamically batch the queries only using the β model. That is to say, in Figure 6, only the first step will be performed. The obtained mini-batches will be directly sent to PIE for execution.

To simulate real-world cases, we assume that the queries’ lengths obey the Gaussian Distribution. Without loss of generality, we set the mean value to 32 and set the standard deviation from 1 to 8. The concurrent tasks are set from 32 to 128. Each task is assigned to a random PET type. For each case, we put 1024 queries in the query pool. For each query in the pool, we randomly assign it to a registered task.

As shown in Figure 12, the proposed CB strategy achieves on average $1.52\times$ and $1.27\times$ speedup over Fixed-Sized Batching and β -only Batching, respectively. When the std values

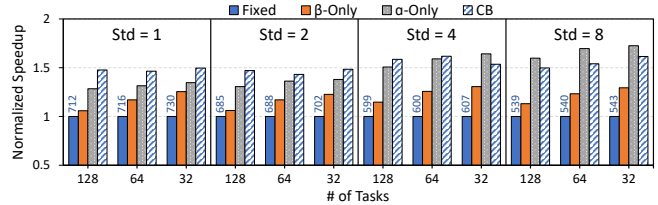


Figure 12: Comparison of Batching Strategies

are set from 1 to 4, CB also achieves up to $1.14\times$ ($1.06\times$ on average) speedup over α -only Batching. For input queries with large variance, Coordinated Batching achieves lower QPS than α -only Batching. We infer that the first step guided by the β model may put some queries with certain length difference into one batch, which is acceptable by Step 1. Such a difference, however, may be amplified in Step 2, since the shared operators usually take up the majority of total execution time (see Figure 9). On the contrary, if the input queries have middle or small variances, the batching of both the two steps is near-optimal. Therefore, to get the highest performance on arbitrary inputs, we can measure the std value of the queries in the pool and choose to use Coordinated Batching (for low-std inputs) or α -only Batching (for high-std inputs).

7 Limitations & Future Work

As revealed by the evaluation results, *PetS* favors the scenarios where the number of tasks is large, while each task has few input queries. If there are only a few tasks, but each task has a large batch, using a traditional SeqS framework may also achieve good throughput. Also, as shown in Figures 8 and 9, when there are only a few tasks, the overhead of *PetS* (i.e., computing PET operators and shared dense operators separately) will outweigh the benefits, since the shared operators cannot achieve enough speedup in these cases to cover the overhead of PET operators.

Currently, the proof-of-concept *PetS* implementation only supports downstream tasks sharing the same pre-trained model. However, with the increasing of tasks adopting transformers as their backbones, there will be more and more shared models registered in *PetS*. The ability to simultaneously serve multiple pre-trained models from various task fields is needed. On the other hand, giant pre-trained models with trillions of parameters exceeding the capability of a single GPU have emerged to achieve significant accuracy gain, such as [13, 52, 63]. Partitioning a single giant model with PET tasks and different shared models on multiple GPUs is challenging for *PetS*. Moreover, for online queries with latency and cost constraints, how to balance *PetS*’s performance and QoS should also be taken into consideration. We leave the model partitioning and QoS-aware query scheduling as *PetS*’s future work.

8 Related Work

Parameter-Efficient Transformers: Apart from the four representative PETs discussed in the paper, there are many

other PET variants like AdapterFusion [41], LoRA [24], LeTs [14] and Prefix-tuning [30], etc. As stated in Section 5.3, these PETs can also work with `PetS` with some necessary extensions. We also notice that there are some works trying to put PETs into one framework such as UniPELT [33] and MAM-Adapter [21]. However, they belong to training frameworks designed from the user’s angle. Their main goal is to combine multiple PET techniques into one model to achieve better accuracy. On the contrary, `PetS` is designed from a service provider’s angle. It batches different fine-tuned tasks provided by users regardless of their PET algorithms and parameters. Therefore, these techniques are orthogonal to `PetS`. Besides, we also notice that Adapter-Hub [42] shares many ideas with `PetS`. It provides an easy way to train and share adapters for different downstream tasks based on Huggingface Transformers. A recent work named OpenDelta [10] further supports fine-tuning more types of PETs. However, the two frameworks are mainly designed for algorithm developers and not optimized for model serving. `PetS` mainly focuses on improving the multi-task serving efficiency from the system implementation and optimization perspectives. We believe that it will be promising to adopt `PetS` as the inference serving backend of these training frameworks.

Inference Serving Systems: As illustrated in Section 6, previous inference serving systems can be classified as `SeqS` and `ParS`. Rafiqi [53] and Clipper [7] deploy a model in an exclusive container, and introduce caching, batching, and model selection techniques to reduce model swapping overhead. Clockwork [19] reduces GPU inference latency variability by ordering queries based on their service level objectives (SLOs) and only running one query at a time, while TurboTransformers [12] batches queries to a single model to improve system throughput. Compared to the `SeqS` running each model sequentially, `ParS` systems enable concurrent execution of multiple models. INFaaS [48] proposes to automatically select models for multiple queries in order to maximize throughput. NVIDIA’s MPS [39] and recent MIG [38] techniques enable efficient GPU resource sharing through hardware partition or full isolation. There exist other systems [8, 40, 60] featured with GPU sharing techniques.

Transformers Inference Engines: With the prevailing of Transformers, some inference engines are designed specifically for efficient Transformer inference. FastTransformers [37] and DeepSpeed [35] are two frameworks featured with multi-GPU inference. LightSeq [56] is a light-weighted inference engine performing some input-aware optimization techniques, such as smart batching and padding minimization, so does the inference engine of TurboTransformers [12].

Leveraging parameter-efficient Transformers, `PetS` saves storage, mitigates model swapping overhead and also improves system throughput by co-design and co-optimization between inference serving system and inference engine.

9 Conclusion

This paper presents `PetS`, a unified framework for efficient multi-task Parameter-Efficient Transformers (PETs) serving. To enable flexible PET task management and high-throughput serving, we first propose a unified representation to put different PETs into the same framework. Then we design a specialized PET inference engine to execute different tasks’ queries in batches. We also propose a coordinated batching strategy to deal with arbitrary input queries and develop a PET operator scheduling strategy to exploit parallelism between PET tasks. Experiments on Edge/Desktop/Server GPUs demonstrate that `PetS` can support up to $26\times$ more concurrent tasks and improves the serving throughput by $1.53\times$ and $1.63\times$ on Desktop and Server GPUs, respectively.

Acknowledgment

We thank all the reviewers and the shepherd for their valuable suggestions. This work is supported by NSF of China (61832020, 62032001, 92064006), Beijing Academy of Artificial Intelligence (BAAI), and 111 Project (B18001)

References

- [1] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.
- [2] Eunjin Baek, Dongup Kwon, and Jangwoo Kim. A Multi-neural Network Acceleration Architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 940–953. IEEE, 2020.
- [3] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Nee-lakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language Models are Few-shot Learners. *arXiv preprint arXiv:2005.14165*, 2020.
- [4] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-End Object Detection with Transformers. *arXiv preprint arXiv:2005.12872*, 2020.
- [5] Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhongyang Wang, and Michael Carbin. The Lottery Ticket Hypothesis for Pre-trained Bert Networks. *arXiv preprint arXiv:2007.12223*, 2020.
- [6] Yujeong Choi and Minsoo Rhu. Prema: A Predictive Multi-task Scheduling Algorithm for Preemptible Neural Processing Units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 220–233. IEEE, 2020.

- [7] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency Online Prediction Serving System. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 613–627. USENIX Association, 2017.
- [8] Abdul Dakkak, Cheng Li, Simon Garcia De Gonzalo, Jinjun Xiong, and Wen-Mei W. Hwu. TrIMS: Transparent and Isolated Model Sharing for Low Latency Deep Learning Inference in Function as a Service Environments. *CoRR*, abs/1811.09732, 2018.
- [9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [10] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. *arXiv preprint arXiv:2203.06904*, 2022.
- [11] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *arXiv preprint arXiv:2010.11929*, 2020.
- [12] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. TurboTransformers: An Efficient GPU Serving System for Transformer Models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.
- [13] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *CoRR*, abs/2101.03961, 2021.
- [14] Cheng Fu, Hanxian Huang, Xinyun Chen, Yuandong Tian, and Jishen Zhao. Learn-to-Share: A Hardware-friendly Transfer Learning Framework Exploiting Computation and Parameter Sharing. In *International Conference on Machine Learning*, pages 3469–3479. PMLR, 2021.
- [15] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU Kernels for Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2020.
- [16] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low Latency RNN Inference with Cellular Batching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys, pages 1–15, 2018.
- [17] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Reddy Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, et al. Planaria: Dynamic Architecture Fission for Spatial Multi-tenant Acceleration of Deep Neural Networks. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 681–697, 2020.
- [18] Google. TensorFlow Serving. <https://github.com/tensorflow/serving>, 2021.
- [19] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 443–462, 2020.
- [20] Demi Guo, Alexander M Rush, and Yoon Kim. Parameter-Efficient Transfer Learning with Diff Pruning. *arXiv preprint arXiv:2012.07463*, 2020.
- [21] Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. Towards a Unified View of Parameter-Efficient Transfer Learning. *CoRR*, abs/2110.04366, 2021.
- [22] Dan Hendrycks and Kevin Gimpel. Gaussian Error Linear Units (GELUs). *arXiv preprint arXiv:1606.08415*, 2016.
- [23] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.
- [24] Edward Hu, Yelong Shen, Phil Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models, 2021.
- [25] Yifan Jiang, Shiyu Chang, and Zhangyang Wang. Transgan: Two Transformers can Make One Strong Gan. *arXiv preprint arXiv:2102.07074*, 1(3), 2021.
- [26] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. Ten Lessons from Three Generations Shaped Google’s TPUv4i: Industrial Product. In *2021 ACM/IEEE 48th*

Annual International Symposium on Computer Architecture (ISCA), pages 1–14. IEEE, 2021.

- [27] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. Heterogeneous Dataflow Accelerators for Multi-DNN Workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 71–83, 2021.
- [28] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A Lite Bert for Self-supervised Learning of Language Representations. *arXiv preprint arXiv:1909.11942*, 2019.
- [29] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 7871–7880. Association for Computational Linguistics, 2020.
- [30] Xiang Lisa Li and Percy Liang. Prefix-Tuning: Optimizing Continuous Prompts for Generation. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 4582–4597. Association for Computational Linguistics, 2021.
- [31] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A Robustly Optimized Bert Pretraining Approach. *arXiv preprint arXiv:1907.11692*, 2019.
- [32] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin Transformer: Hierarchical Vision Transformer Using Shifted Windows. *arXiv preprint arXiv:2103.14030*, 2021.
- [33] Yuning Mao, Lambert Mathias, Rui Hou, Amjad Almahairi, Hao Ma, Jiawei Han, Wen-tau Yih, and Madsian Khabsa. UniPELT: A Unified Framework for Parameter-Efficient Language Model Tuning. *CoRR*, abs/2110.07577, 2021.
- [34] Daniel Mendoza, Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. Interference-Aware Scheduling for Inference Serving. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, page 80–88, 2021.
- [35] Microsoft. DeepSpeed for Inferencing Transformer based Models. <https://www.deepspeed.ai/tutorials/inference-tutorial/>, 2021.
- [36] NVIDIA. CUB. https://nvlabs.github.io/cub/structcub_1_1_caching_device_allocator.html.
- [37] NVIDIA. Fast Transformer. <https://github.com/NVIDIA/FasterTransformer>.
- [38] NVIDIA. MIG. <https://docs.nvidia.com/datacenter/tesla/mig-user-guide/>, 2021.
- [39] NVIDIA. MPS. <https://docs.nvidia.com/deploy/mps/index.html>, 2021.
- [40] NVIDIA. Triton Inference Server. <https://developer.nvidia.com/nvidia-triton-inference-server>, 2021.
- [41] Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. Adapterfusion: Non-destructive Task Composition for Transfer Learning. *arXiv preprint arXiv:2005.00247*, 2020.
- [42] Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. AdapterHub: A framework for adapting transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 46–54, Online, October 2020. Association for Computational Linguistics.
- [43] Sai Prasanna, Anna Rogers, and Anna Rumshisky. When Bert Plays the Lottery, All Tickets are Winning. *arXiv preprint arXiv:2005.00561*, 2020.
- [44] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. Pre-trained Models for Natural Language Processing: A Survey. *Science China Technological Sciences*, pages 1–26, 2020.
- [45] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *OpenAI blog*, 1(8):9, 2019.
- [46] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *arXiv preprint arXiv:1910.10683*, 2019.

- [47] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. A Primer in Bertology: What We Know about How Bert Works. *Transactions of the Association for Computational Linguistics*, 8:842–866, 2020.
- [48] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated Model-less Inference Serving. In *Proceedings of the 2021 USENIX Annual Technical Conference*, pages 397–411, 2021.
- [49] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. *arXiv preprint arXiv:1910.01108*, 2019.
- [50] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A GPU Cluster Engine for Accelerating DNN-based Video Analysis. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 322–337. ACM, 2019.
- [51] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- [52] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-billion Parameter Language Models using Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- [53] Sammy Sidhu, Jordon Wing, and Aakash Japi. Rafiqi: A GPU-Based Deep Learning Model Serving System. Technical report, University of California, Berkeley, 2020.
- [54] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. GLUE: A Multi-task Benchmark and Analysis Platform for Natural Language Understanding. *arXiv preprint arXiv:1804.07461*, 2018.
- [55] Wenhai Wang, Enze Xie, Xiang Li, Deng-Ping Fan, Kaitao Song, Ding Liang, Tong Lu, Ping Luo, and Ling Shao. Pyramid Vision Transformer: A Versatile Backbone for Dense Prediction without Convolutions. *arXiv preprint arXiv:2102.12122*, 2021.
- [56] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. LightSeq: A High Performance Inference Library for Transformers. *arXiv preprint arXiv:2010.13887*, 2020.
- [57] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, 2009.
- [58] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45. Online, October 2020. Association for Computational Linguistics.
- [59] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. XLnet: Generalized Autoregressive Pretraining for Language Understanding. In *Advances in neural information processing systems*, pages 5753–5763, 2019.
- [60] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. *CoRR*, abs/1902.04610, 2019.
- [61] Li Yuan, Yunpeng Chen, Tao Wang, Weihao Yu, Yujun Shi, Zihang Jiang, Francis EH Tay, Jiashi Feng, and Shuicheng Yan. Tokens-to-token vit: Training Vision Transformers from Scratch on Imagenet. *arXiv preprint arXiv:2101.11986*, 2021.
- [62] Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models. *arXiv preprint arXiv:2106.10199*, 2021.
- [63] Wei Zeng, Xiaozhe Ren, Teng Su, Hui Wang, Yi Liao, Zhiwei Wang, Xin Jiang, ZhenZhang Yang, Kaisheng Wang, Xiaoda Zhang, Chen Li, Ziyang Gong, Yifan Yao, Xinjing Huang, Jun Wang, Jianfeng Yu, Qi Guo, Yue Yu, Yan Zhang, Jin Wang, Hengtao Tao, Dasen Yan, Zexuan Yi, Fang Peng, Fangqing Jiang, Han Zhang, Lingfeng Deng, Yehong Zhang, Zhe Lin, Chao Zhang, Shaojie Zhang, Mingyue Guo, Shanzhi Gu, Gaojun Fan, Yaowei Wang, Xuefeng Jin, Qun Liu, and Yonghong Tian. PanGu- α : Large-scale Autoregressive Pretrained Chinese Language Models with Auto-parallel Computation. *CoRR*, abs/2104.12369, 2021.
- [64] Mengjie Zhao, Tao Lin, Fei Mi, Martin Jaggi, and Hinrich Schütze. Masking as an Efficient Alternative to Finetuning for Pretrained Language Models. *arXiv preprint arXiv:2004.12406*, 2020.

A Artifact Appendix

Abstract

The artifact contains PetS’s code and its setup&running descriptions. We provide instructions and click-to-run scripts for reproducing the main results in this paper.

Scope

This artifact is used for reproducing the main results in Section 6. Specifically, we produce click-to-run scripts to reproduce the results of Figures 7,8,9,11,12 and Table 4.

Contents

- **Code Base:** The code base of the artifact includes the PetS inference framework. It contains the coordinated batching and PET operator scheduling components to demonstrate PetS’s performance optimization strategies.
- **Benchmarking Scripts:** We provide click-to-run benchmarking scripts to evaluate PetS’s performance. The script `run_pets_main_results.sh` can conduct all the main experiments, while you can also run each experiment individually using other provided scripts like `eval_batching_strategies.sh`, `eval_multi_stream.sh`, etc.
- **Instructions:** We provide a detailed README to guide the environment setup, evaluation and code reuse, etc.
- **Reference Results:** We provide the experiment results on two GPU platforms for reference use.

Hosting

The [†] artifact is archived in Zenodo.

Requirements

- **Hardware:** The artifact can run on a server or embedded platform equipped with at least one NVIDIA GPU. We tested NVIDIA Jetson TX2, GeForce GTX 1080 Ti, and NVIDIA Tesla V100.
- **Compilation and Runtime:** The experiments are performed on three platforms with GPUs mentioned above. The compilers and operating systems on the platforms are: on platform with 1080 Ti GPU, g++ 7.5.0 and nvcc 11.3, Ubuntu 20.04; Tx2: Jetpack 4.4.1 with CUDA-10.2. V100: Ubuntu 18.04, CUDA-10.1. g++ 7.5.0.

Many other GPU platforms (e.g., 2080Ti, P100, K80, etc.) may also be compatible with this artifact. However, the Ampere architecture (e.g., A100, A6000) is not currently supported by the sputnik library.

[†]<https://doi.org/10.5281/zenodo.6534753>

Evaluation and Expected Results

After setting up the environment, you can run the two-step evaluation procedure: experiments running and results validating. The first step generates the performance numbers, and the second step draws figures. Please refer to `README.md` for detailed evaluation flow. The full evaluation lasts for about one hour. You can find the plotted results in the `research/reproduced_figures` folder.

Known Issues: Some AE reviewers have reported that in their running environments with V100 GPUs, they failed to get the same curve as Figure 11 (though Figure 11 is based on the 1080-ti GPU). We tested two machines with V100 GPUs. One can get even better results (a local machine, driver version = 510), but the other (an AliCloud instance, driver version = 460) got worse results than 1080-ti. We infer that this is due to the hardware and driver differences. The exact cause is still under investigation.

How to Reuse Beyond Paper

A PET algorithm can work with PetS as long as it meets two requirements:

- Its PET operations are separable (with necessary equivalent-transformations) from the shared operations.
- The separated PET operations are light-weighted.

To support a new algorithm, we should first identify its PET operations. Then three steps are required to add the new PET algorithm to PetS:

- Step-1. Register a new PET type and implement the PET operations using Pytorch APIs in `python/turbo_transformers/layers/modeling_pets.py`.
- Step-2. Deal with the PET parameters loading. Add new loading functions in `modeling_shared_bert.py` and `pet_manager.h`, respectively.
- Step-3. Implement the new PET operators in `shadow_op.cpp/shadow_op.h`
- Step-4. If the new PET operators should be called at places that are different from the four PETs in the paper, you should also modify the bert layers backends, e.g., `bert_output.cpp`

Campo: Cost-Aware Performance Optimization for Mixed-Precision Neural Network Training

Xin He

CSEE, Hunan University & Xidian University

Jianhua Sun

CSEE, Hunan University

Hao Chen

CSEE, Hunan University

Dong Li

University of California, Merced

Abstract

Mixed precision training uses a mixture of full and lower precisions for neural network (NN) training. Applying mixed precision must cast tensors in NN from float32 (FP32) to float16 (FP16) or vice versa. The existing strategy greedily applies FP16 to performance-critical operations without quantifying and considering the casting cost. However, we reveal that the casting cost can take more than 21% of NN operation execution time, and in some cases surpasses the performance benefit of using low precision. In this paper, we introduce Campo, a tool that improves performance of mixed-precision NN training with the awareness of casting costs. Campo is built upon performance modeling that predicts the casting cost and operation performance with low precision, and introduces a cost-aware graph rewriting strategy. Campo is user-transparent, and enables high performance NN training using mixed precision without training accuracy loss. Evaluating Campo with six NN models, we show that compared to TensorFlow using TF_AMP (a state-of-the-art performance optimizer for mixed precision training from Nvidia), Campo improves training throughput by 20.8% on average (up to 24.5%) on RTX 2080 Ti GPU and by 20.9% on average (up to 23.4%) on V100 GPU, without training accuracy loss. Because of using the cost-aware mixed precision training, Campo also improves energy efficiency by 21.4% on average (up to 24.2%), compared to TensorFlow using TF_AMP.

1 Introduction

Training Neural network (NN) can be resource-demanding: it consumes many compute cycles and requires high memory bandwidth and capacity. One promising approach to lower the resource requirements is to use mixed precision training [1]. Mixed precision is a computational method using a mixture of full and lower precisions. Mixed precision can deliver significant computational speedup by executing operations in a lower precision format as much as possible, while storing critical information in the full-precision format to preserve task-specific accuracy. The mixed precision training

has shown significant speedup over the single (full) precision training on a variety of NN [2–4], especially with tensor cores (TC) available on some GPU architectures.

In NN training frameworks like TensorFlow and PyTorch, the mixed precision training is implemented via a graph rewrite process [5]. This process casts tensors referenced in certain operations in NN from float32 (FP32) to float16 (FP16), or vice versa. This process adopts a greedy strategy that applies FP16 execution to performance-critical operations, most of which are matrix multiplication and convolution, based on an implicit assumption that using low precision *always* leads to performance improvement, and hence the casting cost can be always justified.

However, our detailed performance analysis reveals that the above common assumption is not true. We observe that the casting cost can take more than 21% of operation execution time, depending on the input tensor size of the operation using mixed precision. In some cases (e.g., the operation `MatMul` with an input data size of (64, 1001, 1001, 2048)), the casting cost surpasses the performance benefit of using low precision. As a result, using mixed precision training (even with TC) may not be performance-beneficial and even leads to performance loss (22.7% in the example of `MatMul`). Hence, the casting cost must be considered and quantified when deciding precision for operations.

Deciding whether using low precision for an operation is performance-beneficial is challenging, because the operation time and casting cost are affected by input data size, whether TC is used, and performance characterization of operation (e.g., memory access pattern and compute intensity). Also, using low precision should not impact NN model accuracy. Hence, making the decision of using low precision is a multi-dimensional problem.

In this paper, we introduce a cost-aware performance optimization tool, named Campo, aiming to improve performance of mixed-precision NN training with the awareness of casting costs. Campo assigns the low or full computation precision to each operation in NN to maximize performance while preserving the NN model accuracy. Campo is built upon perfor-

mance modeling that predicts the casting cost and operation performance with low precision. The performance modeling is operation-specific and captures events (such as L2 cache misses and global loads/stores on GPU) critical to the performance of low precision and collected through dynamic profiling in full precision. Using dynamic profiling, the performance modeling is also able to capture the impact of input data size on performance.

Leveraging the performance modeling, Campo introduces a cost-aware graph rewriting strategy. This strategy avoids applying low precision to those operations that cannot get performance benefit, and minimizes the casting cost when applying low precision to a group of operations. Furthermore, Campo does not impact NN model accuracy, because it only applies low precision to those operations identified as numerical safe by the traditional algorithm for low precision assignment. In addition, some operations can benefit from low precision but cannot run on TC because their input data sizes cannot meet the requirement of TC. For those operations, Campo pads the input tensors without programmer participation to maximize the utilization of TC for high performance.

We summarize major contributions as follows.

- We conduct a comprehensive performance characterization on operations in NN training and quantify casting costs. In contrast to the traditional methods that decide precision assignment without considering the casting cost, we reveal that the casting cost can outweigh the performance benefit of using low precision. This observation is unprecedented.
- We develop novel and practical performance modeling to predict casting cost and the performance of operations in low precision.
- We propose Campo, a performance optimization tool that enables high-performance mixed precision training without losing training accuracy. Campo uses a graph traverse algorithm and performance modeling to assign low or high precision to each operation.
- We implement Campo within TensorFlow, and evaluate it with six NN models on Nvidia GeForce RTX 2080 Ti and V100 GPUs. Our evaluation shows that compared to TensorFlow using TF_AMP (a state-of-the-art performance optimizer for mixed precision training from Nvidia), Campo improves training throughput by 20.8% on average (up to 24.5%) on RTX 2080 Ti and by 20.9% on average (up to 23.4%) on V100, without losing training accuracy. Because of using cost-aware mixed precision training, Campo improves energy efficiency by 21.4% on average (up to 24.2%), compared with TensorFlow using TF_AMP.

2 Background

2.1 Mixed Precision Training

A dominant programming paradigm, commonly adopted by machine learning frameworks such as TensorFlow and Py-

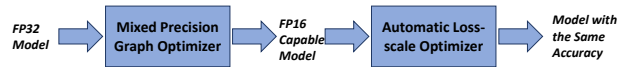


Figure 1: The workflow of mixed precision training.

Torch, is to represent an NN model as a static dataflow graph, where computation functions (e.g., Conv2D, and MatMul) in the NN model are associated with nodes in the graph, and input and output tensors of the computation map to edges. The architecture of the NN model (i.e., the dataflow graph) is defined using symbolic expressions by the programmer; The common computation functions are defined as *operations* by the machine learning framework.

The mixed precision training makes precision assignment decisions per node. Some nodes (i.e., operations) in the dataflow graph use full precision (i.e., FP32), and those nodes are essential to maintain training accuracy. Other nodes use lower precision, which is useful to save memory capacity and bandwidth and enable faster math operations (especially on GPUs with TC support). As a result, the mixed precision training can harvest the best of both worlds: maintaining training accuracy and having fast execution.

In TensorFlow and PyTorch, only FP16 is considered as lower precision for the mixed precision training because of FP16’s commonality in various GPU architectures, although some GPU architectures (such as Turing and Ampere) support other lower precisions, such as INT8 and INT4. Like TensorFlow and PyTorch, we only consider FP16 as lower precision in this paper, because of FP16’s commonality.

Figure 1 depicts the workflow of using the mixed precision training in TensorFlow and PyTorch. In general, it includes two steps: (1) identifying which nodes should be changed to FP16 and inserting casts between FP32 nodes and FP16 nodes by a mixed precision graph optimizer, and (2) adding loss scaling to preserve small gradient values by an automatic loss-scale optimizer. We focus on (1) in this paper.

The mixed precision graph optimizer decides the assignment of low precision to operations by classifying operations into multiple lists based on operation’s numerical safety. The numerical safety refers to how an NN model’s quality is affected by the use of low precision. An operation is numerical unsafe, if using low precision during the operation execution leads to worse training accuracy, compared with using FP32. In TensorFlow, there are four lists, discussed as followed.

- **Allowlist:** operations (e.g., MatMul and Conv2D) in this list are considered numerically-safe for execution in FP16, and also performance-critical. These operations are always converted to use FP16.
- **Denylist:** operations (e.g., Exp and SoftMax) in this list are considered numerically-dangerous in FP16 and their effects may also be observed in a downstream operation node. For example, using FP16 in the operation sequence of Exp ->Add, the Add is unsafe due to the unsafe Exp.

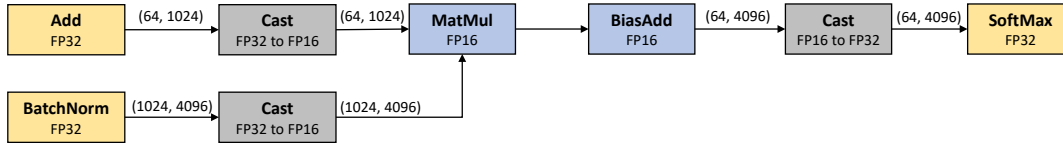


Figure 2: A snippet of the dataflow graph from BERT using mixed precision. The tuple on each edge represents a tensor with its shape (i.e., the size of each dimension).

- **Inferlist:** operations (e.g., `BiasAdd`) in this list are considered numerically-safe in FP16, which may be, however, made unsafe by an upstream *denylist* operation.
- **Clearlist:** operations (e.g., `Max` and `Min`) in this list do not have numerically-significant effects in the sense that they can be executed either in FP16 or in FP32.

Compared with TensorFlow, PyTorch use three lists because the operations in the *clearlist* and the *inferlist* are classified into a single list.

Using the above lists, the mixed precision graph optimizer uses low precision for an operation, if any of the following three conditions is true: (1) The operation is in the *allowlist*; (2) the operation is in the *clearlist*, and its immediate ancestor(s) and immediate descendent(s) are using low precision; (3) the operation is in the *inferlist* and there is no upstream *denylist* operation. The mixed precision graph optimization works online by re-writing the dataflow graph via inserting casts before the iterative training of NN model happens.

Figure 2 shows a mixed-precision graph snippet taken from BERT [6] (a transformer-based model). To enable FP16 for `MatMul` and `BiasAdd` while ensuring numerical safety, two FP32-to-FP16 cast operation nodes and one FP16-to-FP32 cast operation node are inserted into the graph. Take the FP32-to-FP16 cast operation node with the input data size (64, 1024) as an example. The number of scalar elements in the input data (tensor) is 65536 (i.e., 64×1024) in this example.

2.2 Tensor Core Acceleration

Since Volta architecture, Nvidia introduces specialized hardware arithmetic units into its GPU products, called Tensor Cores (TC). Compared to regular CUDA cores, TC is more performant and energy-efficient. TC is used to accelerate FP16 matrix multiplication and convolution operations. In TensorFlow, these operations refer to `MatMul`, `Conv2DBackpropFilter`, `Conv2DBackpropInput` and `Conv2D`, which are usually the most fundamental and time-consuming operations in NN models.

TC is automatically activated to run an operation when two conditions are met: (1) the operation is either matrix multiplication or convolution using FP16, and (2) the input tensors of the operation satisfy the shape requirements. For (2), TC requires certain dimensions of the tensor to be a multiple of 8. If the condition (1) is met, we say the operation is a *TC candidate*. Such an operation can run on regular CUDA cores

with low precision when the condition (2) is not met.

Besides the above discussion on the mixed precision training and TC, we target on those NN models whose dataflow graphs are static, which indicates that the dataflow graph does not change its structure across training samples and hence each training step goes through the exactly same computation graph. This implies that once the training batch size is determined, the input data size and shape (i.e., the size of each dimension) of operations are known before the training happens. Such NN models are very common and have been a research target in many recent efforts [7–13].

3 Observation and Motivation

To motivate the design of Campo, we characterize the performance of operations under full precision and low precision, and study the casting cost. Table 1 shows the performance results for six operations. These operations can be commonly found in NN models. The first four operations in the table (`MatMul`, `Conv2D`, `Conv2DBackpropFilter`, and `Conv2DBackpropInput`) are candidates to run on TC. These four operations can easily account for most of the NN training time. For example, in ResNet50, the four operations take more than 90% of the total training time. The four operations fall into the *allowlist*, and hence are always converted for FP16 execution. The last two operations in the table (i.e., `BiasAdd` and `MaxPool`) fall into the *inferlist* and *clearlist*, respectively, of which the numerical precision chosen for the operations is usually context-dependent. We do not study the operations in the *denylist* because they are always executed in FP32. Besides the six operations in Table 1, we study other operations in the three lists, but do not show them in Table 1 for brevity.

We develop two microbenchmarks for each operation to run it with FP16 and FP32 respectively. The input data sizes for each operation are collected from Resnet-50, Inception3 and DCGAN by dlprof [14]. Among those input data sizes, some of them meet the TC requirement on tensor shape, and hence the corresponding operations run on TC. In our study, we use TensorFlow 1.15 and Nvidia RTX 2080 Ti GPU. We run each operation with each input data size 100 times and report the average result. In Table 1, “FP16 Exe. time” and “FP32 Exe. time” do not include casting cost.

Overall, we study the impact of data precision, TC, casting cost, and input data size on operation performance.

Table 1: Performance comparison of some of the representative operations in NN training.

NN Operations	Input Data Size	FP16 Exe. Time (ms)	FP16+Cast Exe. Time (ms)	FP32 Exe. Time (ms)	Using TC
MatMul	(2048, 8, 8, 1024)	0.312	0.353	0.323	yes
	(64, 1001, 1001, 2048)	0.412	0.524	0.427	no
	(2048, 1024, 1024, 1024)	0.414	0.584	0.888	yes
Conv2D	(64, 35, 35, 48)	2.707	2.795	3.664	yes
	(64, 147, 147, 32)	28.965	29.249	29.487	no
	(64, 299, 299, 3)	57.879	58.944	60.098	no
Conv2DBackpropFilter	(64, 299, 299, 3)	8.690	9.773	10.246	no
	(64, 149, 149, 32)	6.013	7.988	7.011	no
	(64, 35, 35, 192)	0.786	0.948	0.871	yes
Conv2DBackpropInput	(64, 37, 37, 96)	3.954	4.049	6.943	yes
	(64, 149, 149, 32)	15.561	16.828	15.696	no
	(64, 35, 35, 192)	5.234	5.939	10.060	yes
BiasAdd	(64, 1001, 1001)	0.252	0.317	0.255	no
	(64, 4096, 4096)	0.294	0.323	0.298	no
	(64, 9216, 9216)	0.299	0.342	0.311	no
MaxPool	(64, 35, 35, 288)	1.849	2.072	1.793	no
	(64, 17, 17, 768)	1.399	1.542	1.402	no
	(64, 8, 8, 2048)	0.825	1.128	0.981	no

1) *Performance variance with different data precisions.*

Table 1 shows that across operations, FP16 consistently outperforms F32, regardless of using TC or not (and without consideration of casting cost). For example, despite not using TC, MatMul with FP16 performs slightly better (3.5%) than with FP32 for the input data size (64, 1001, 1001, 2048).

Furthermore, when the input shape meets the requirement of using TC, the performance gain of using FP16 over FP32 is more significant. For example, Conv2DBackpropInput with FP16 performs significantly better (48%) than with FP32 for the input data size (64, 35, 35, 192).

Observation 1. Without TC, using F16 leads to slightly better performance than using F32. Using TC for FP16 magnifies the performance benefit of F16.

2) *Impact of input data size on performance gains from FP16.* Training an NN model can invoke many instances of an operation in a training step. Different instances of the operation can use different input data sizes. Table 1 shows that as we change the input data size of an operation, the performance gain of using FP16 over using FP32 varies significantly. For example, for MatMul with FP16, the performance gain is 3.6% and 114.5% for the input data sizes (64, 1001, 1001, 2048) and (2048, 1024, 1024, 1024) respectively. In this example, such a large performance variance comes from whether TC is utilized. Even if TC is not utilized for TC candidate operations, we observe large performance variance across different input data sizes. For example, Conv2DBackpropInput with input data sizes (64, 149, 149, 32) and (64, 37, 37, 96) have 75.6% and 0.9% performance gains when using FP16.

The above observation holds true for the non-TC candidate operations as well. For example, BiasAdd with input data sizes (64, 9216, 9216) and (64, 1001, 1001) have 40.1% and 11.9% performance gain when using FP16.

The reason for the above result is because of smaller memory bandwidth consumption and smaller number of FP operations with smaller input data size, which offers less opportunity for FP16 to tap and improve performance.

Observation 2. The performance gain of using FP16 varies largely across input data sizes.

3) *Impact of casting cost.* Comparing “FP16+Cast Exe. Time” and “FP16 Exe. Time” in Table 1, we see that the cast operation introduces 3% - 29% overhead, diminishing the performance benefit of FP16. As a result, using FP16 can perform worse than FP32. For example, considering the casting cost, MatMul using a TC-satisfied input data size (2048, 8, 8, 1024) and a TC-unsatisfied input data size (64, 1001, 1001, 2048) with FP16 performs worse than with FP32 by 9.3% and 22.7% respectively, and the casting cost takes 11.6% and 21.4% of the operation execution time, which is large.

The casting cost stems from (1) the time to initialize the cast operation node in the dataflow graph, and (2) the time to do bitcast and numerical truncation for each scalar element in the input tensor as well as construct the output.

Observation 3. The cast operation introduces non-negligible overhead. Considering the casting cost, it is not always performance-profitable to convert FP32 to FP16 regardless of using TC or not.

In addition to the NVIDIA GeForce RTX 2080 Ti, we get the same three observations on Nvidia V100.

Implications of observations. The effectiveness of using low precision for an operation is impacted by input data size, casting cost, and the usage of TC. Optimizing the assignment of low precision to operations is a multi-dimensional problem, not just one dimensional problem as assumed in the existing solutions.

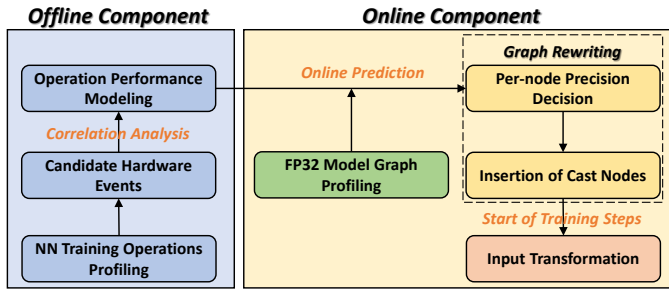


Figure 3: Overview of Campo.

4 Design

4.1 Overview

Campo includes an offline component and an online component, as illustrated in Figure 3. The offline component is used to build performance modeling to predict performance of operations in FP16. The performance modeling is used by the online component, and makes performance prediction using performance events (e.g., L2 cache misses and global memory load/store throughput) collected from operation execution in FP32. The performance modeling uses correlation analysis to decide which events are the most important for accurate performance prediction. The performance modeling is based on statistical regression modeling, which is built only once by the offline component but repeatedly used by the online component.

The online component includes graph profiling, graph traverse to make per-node precision decision, insertion of cast operation, and input transformation. The graph profiling uses one iteration to run operation nodes in FP32 and collect events needed by performance modeling. The graph traverse makes four passes on the dataflow graph of the NN model and uses performance modeling to decide if each operation should use FP16 or not based on the estimation of performance benefit and cost of using FP16. The input transformation pads the input tensors to make TC candidates meet the requirement of TC on input shape and thus improves the utilization of TC. The online component is lightweight and enables cost-aware mixed precision optimization with guarantee on performance improvement without training accuracy loss.

4.2 Performance Modeling

We build performance modeling to decide whether low or full precision should be employed for a given operation with a given input data size. The performance modeling predicts (1) the casting cost based on the input data size, and (2) execution time of the operation with low precision.

4.2.1 Predicting Casting Cost

The casting cost for a cast operation includes two parts: (1) the time to initialize the cast operation node in the dataflow graph, denoted by C_I , and (2) the time to do the conversion (i.e., the cost to do `bitcast` and numerical truncation for each scalar element in the input tensor as well as construct the output tensor), denoted by C_C .

C_I is modeled as a constant, because C_I comes from a couple of memory allocations and variable assignments for the object initialization of the cast operation node in the dataflow graph. C_C is proportional to the number of scalar elements in the input tensor of the cast operation node. This proportion is represented as a ratio, r . Given the same number of scalar elements, we observe that there is no performance difference between converting from FP16 to FP32 and from FP32 to FP16.

Hence, the casting cost is modeled in Equation 1, where $tensor_size$ is the input data size of the cast operation node. Taking the operation `Add` with an input data size (64, 1024) for conversion as an example, $tensor_size$ is 65536 (i.e., 64×1024).

$$casting_cost = r * tensor_size + C_I \quad (1)$$

To use the above model, we must know r and C_I . They are obtained using linear regression where r and C_I are the slope coefficient and intercept. In particular, by profiling 500 FP32-to-FP16 cast cases and 500 FP16-to-FP32 cast cases, we collect a total of 1000 training samples, each of which includes a pair of measured $casting_cost$ and $tensor_size$ from a cast operation. We use the method of least squares to find the values of r and C_I that minimize the sum of the squared errors.

4.2.2 Predicting Execution Time of Operation

We adopt an operation-specific modeling method, which means that we build a performance model for each individual operation. We do not build a general model for all the operations to predict performance, because the operations exhibit a variety of performance characteristics in terms of memory access patterns and computation intensity. Building a single, general model does not give good prediction accuracy. In our experience, we build a general model using the similar method as building operation-specific models, but can achieve only 43% prediction accuracy, which leads to 21% longer execution time, compared with using operation-specific models. Furthermore, although the total number of operations in the *allowlist*, *inferlist*, and *clearlist* is 143, which is large, the operation-specific performance models are built offline for once, and then can be reused for all NN models. Hence, operation-specific modeling is practical.

Why not using dynamic profiling to measure performance in low precision? Our performance modeling predicts

the execution time of an operation using low precision on regular CUDA cores and TC (if the operation is an TC candidate). We could use dynamic profiling to measure the execution time. In particular, we use three training iterations of the NN model: one iteration running all operation nodes in FP32, one running all operation nodes in FP16 on regular CUDA cores, and one iteration running TC candidate nodes in FP16 on TC.

However, the dynamic profiling using training iterations has limitations. In particular, the dynamic profiling uses FP16 for many operations to measure execution time, regardless of the impact of those operations on training convergence. To avoid slow convergence of the training process after dynamic profiling, one has to discard the three training iterations and restart the training process. This complicates the training pipeline. Even worse, for those downstream training tasks that use a pre-trained model, the number of training iterations is limited. Using dynamic profiling can lead to a large increase in training time. For example, training (fine-tuning) a small NN model Audio2Vec [15] can take about ten iterations, and losing three iterations for dynamic profiling leads to more than 20% increase in training time.

Modeling intuition. The performance of an operation in FP32 and FP16 has correlation. For example, using FP16 reduces the working set size, compared with using FP32, which can lead to less cache misses and in turn decrease execution time. Our intuition is that using execution time in FP32 and a handful of performance-critical events measured in FP32, we can predict execution time in FP16. The parameters (coefficients) in our performance modeling should capture the performance correlation between FP32 and FP16.

Performance modeling. Based on the above intuition, we build the performance model as follows.

$$OP_{LP} = OP_{FP32} \cdot \left(\sum_{i=1}^N w_i \cdot PC_i \right) + \sigma \quad (2)$$

where OP_{LP} is the execution time on either regular CUDA cores or TC using FP16, OP_{FP32} is the execution time on regular CUDA cores using FP32, PC_i is a performance-critical event measured during the execution of FP32, N is the number of performance critical events, and w_i and σ are coefficients. Each operation has up to two performance models: one for regular CUDA cores and the other for TC.

A performance-critical event is a model feature. We use hardware performance counters on GPU to collect those events. There are about 100-200 events collectable on GPU. We choose those events that are the most correlated to the operation performance in FP16, using the following method.

Selection of model features. We use the Spearman's rank correlation coefficient [16] (or Spearman's ρ) to select events. The Spearman's ρ is a method to quantify how well the relationship between two variables can be described using a monotonic function [16]. The Spearman correlation between two variables is high when observations have a similar rank

between the two variables, and low when observations have a dissimilar rank between the two variables.

In our case, an event is a variable and the operation performance using FP16 is the other variable. We make many observations by running tests. If the observations on the event and the observations on the F16 performance have the similar rank (i.e., relative position label of the observations within the event or F16 performance), then the event is monotonically correlated (either monotonically increasing or monotonically decreasing) with the FP16 performance. In our case, we use a threshold of 0.75 for ρ . When $|\rho|$ for an event is larger than 0.75, then we choose it as a model feature.

In particular, given an operation OP , we use the following method to select features to build the performance model for FP16 on TC. Using 1000 different TC-satisfied inputs, we run OP in FP32 1000 times to collect execution time and each collectable event. Then, using the same 1000 inputs, we run OP in FP16 1000 times to collect execution time. As such, we construct 1000 samples, each of which consists of measured FP32 execution time, the value of each collectable event, and FP16 execution time. For each event, we use the 1000 samples to run the Spearman's rank correlation analysis and calculate ρ . If ρ is larger than the threshold, then the event is selected.

To select the features to build the performance model for FP16 on regular CUDA cores, we use the same method as above but the operation inputs do not necessarily meet the TC requirement.

We select events for each operation using the above approach. We discuss the most common events across operations as follows.

- **Global memory load/store throughput** indicates intensiveness of global memory access. If an operation has higher throughput in global memory load/store, the operation may get larger performance benefit by using FP16.
- **Instruction executed per cycle** indicates compute intensity. Using FP16 can be helpful for those floating point intensive operations, because of higher throughput of FP16 instructions.
- **GPU occupancy** indicates how many warps are able to be active during operation execution. An operation with low GPU occupancy will be sensitive to whether FP16 or FP32 is used, because using FP16 or not causes difference in global memory accesses and the operation with low GPU occupancy has lower thread-level parallelism to hide long global-memory access latency.
- **L2 cache accesses and misses and L1 cache accesses** indicate memory access locality in the operation. Compared with an operation with bad reference locality, an operation with good reference locality can take advantage of the cache hierarchy and is not sensitive to global memory bandwidth, hence less sensitive to the global memory bandwidth savings due to the use of FP16.

Getting model coefficients w_i and σ . For each operation-specific performance model, we run the operation 1000 times

with 1000 inputs with different sizes, using FP32 and FP16 respectively. That generates 1000 samples, each of which includes FP32 execution time, the values of events collected in FP32, and FP16 execution time. Using the 1000 samples, we use the method of least squares to find the values of coefficients w_i and σ that minimize the sum of the squared errors. To generate 1000 inputs with different sizes, we profile the input data size of the operations from 11 NN models (including GoogLeNet, UNet-3D, DLRM, DCIGN, BiLSTM, SSD-MobileNet-v1, ShuffleNet, SSD, DenseNet, Mask R-CNN, RNN-T) from the MLPerf benchmark suite [17] using 100 training steps with various batch sizes.

Building performance models for an operation is not time-consuming. For example, building the two models for the operation `Conv2D` (including generating samples to get the model coefficients) takes about 1.5 hours. Building performance models for 143 operations takes about 112.5 hours.

Justification of modeling method. In essence, our modeling method is linear regression. Before we used it, we asked if other modeling methods (such as using a machine learning model) can work. We built a multilayer perceptron model (MLP) taking the same input and output as our linear regression model. The MLP has four layers (one input, one output and two hidden layers) and has 800 neurons in total. However, we do not see any benefit of using such a model in terms of prediction accuracy: the prediction accuracy for the MLP is 71%, while it is 94.2% for the linear regression model. The low prediction accuracy of using MLP is largely due to the fact that our problem nature exhibits near-linear correlation between features and MLP seems to be prone to get stuck in a local optimum for this problem. Furthermore, the MLP takes 10x more training samples than the linear regression model. Hence, we do not use MLP.

Furthermore, we asked if basic heuristics can work. In fact, compared with using dynamic profiling (a basic heuristic), using performance modeling reduces training time by 20% for `Audio2Vec`. Compared with using the same precision for all instances of each operation (another basic heuristic), using performance modeling reduces training time by 35% for `BERT-large`. Our performance models are repeatedly used for NN models, which amortizes the construction cost.

Our modeling method considers the impact of operation input on operation performance, because it uses dynamic profiling in FP32 to measure performance with the given operation input, based on which to make prediction for FP16 performance with the same operation input.

Furthermore, our modeling method is operation-specific, which greatly simplifies model construction, because the operation type itself provides much of implicit information on operation characteristics. For example, the operation name `MatMul` indicates strided memory accesses, and hence the operation-specific performance model does not need to explicitly model such a memory access pattern to make performance prediction. As a result, our performance model can

focus on capturing the correlation between the performance of FP32 and FP16.

4.3 Runtime Graph Rewriting

The runtime graph rewriting decides (1) data precision for each operation, and (2) which operations to be converted together to reduce the number of cast operation nodes. By graph rewriting, Campo aims to reach the following goals: (1) minimizing the training time; (2) minimizing the casting cost; and (3) no adverse impact on the numerical safety (compared with the traditional mixed-precision training).

The graph rewriting in Campo includes graph profiling, graph traverse to determine the precision assignments, and insertion of cast operation nodes, discussed as follows.

Graph profiling. Given an NN model, Campo uses a single training step (or iteration) running in FP32 to collect execution time and events needed by performance modeling for those operations in *allowlist*, *inferlist*, and *clearlist*. The graph profiling is triggered right after the first few training steps used by TensorFlow for warmup (i.e., determining system configurations).

Graph traverse happens after graph profiling, and performs four times on the dataflow graph. Each graph traverse follows the data flow in the NN training to analyze operation nodes in the dataflow graph. During the graph traverses, Campo uses two lists, *allow_nodes* and *deny_nodes*, to record those operation nodes determined to run in FP16 and in FP32 respectively. We depict the four-time traverse as follows.

Traverse #1. During the traverse, when an operation node is encountered, Campo checks if it is in *allowlist*. If yes, then Campo uses performance modeling to decide if the casting cost plus FP16 execution time of the operation (on regular CUDA cores or TC) is smaller than the counterpart FP32 execution time. If yes, then the operation node is put into *allow_nodes*; If no, then it is put into *deny_nodes*. During the traverse, only those operations that are numerical safe in FP16 (i.e., in *allowlist*) are considered, in order to maintain the training accuracy of the NN model.

Traverse #2. During this traverse, Campo checks the remaining operation nodes. For each node, Campo checks if it is either numerically-unsafe (i.e., in *denylist*) or on a path from a node in *denylist* to another node in *denylist* or *inferlist* through some operation nodes in *inferlist* or *clearlist*. If yes, then the checking node is added to *deny_nodes*. This traverse aims to prevent numerically-unsafe operation nodes and their downstream operation nodes from being changed to FP16, in order to maintain the training accuracy.

Traverse #3. During this traverse, Campo checks each remaining operation node. If the node (called the target node in the remaining discussion) is in *inferlist* or *clearlist*, then Campo put the target node into *allow_nodes*. After Traverse 2, such a node should be safe to use FP16. It is possible that the immediate upstream or downstream node(s) of the target

node is in *allow_nodes*. For such a case, the cast operation to convert target node for FP16 or FP32 is saved for higher performance.

Traverse #4. During this traverse, Campo checks each remaining operation node. If the node (called the target node in the remaining discussion) is in *clearlist* and connected to a node in the *allow_nodes* via other nodes in *clearlist*, then Campo uses performance modeling to decide whether the casting cost is smaller than the performance benefit of using FP16 (on regular CUDA cores or TC) for the target node and other connecting nodes. If yes, then the target node is put into *allow_nodes*.

Insertion of cast operation nodes. After the four-time graph traverse, Campo changes the type attribute of operation nodes according to their FP16 or F32 assignments, and then inserts a cast operation node at the boundary between any FP16 node and its neighbour FP32 node (or vice versa) by using the API `ChangeTypeAttrsAndAddCasts` provided by TensorFlow.

4.4 Usage of Tensor Cores

For any operation node in FP16 decided in the graph rewriting process, Campo is able to use performance modeling to decide if using regular CUDA cores or TC is more performance beneficial. If using TC is better, then Campo makes the best efforts to run the operation on TC.

In particular, in each training step, Campo checks the input shape of each TC candidate. If the input shape of a TC candidate cannot meet the TC requirements, Campo pads the input tensor by adding zero-filled rows or columns. For example, for `Conv2D`, Campo pads its input to make the dimension size of each channel a multiple of 8. Compared to the traditional padding method recommended by `dlprof`, our method is implemented inside the training framework, and thus transparent to the users and does not need to modify NN models.

Overhead analysis. Zero padding adds overhead to computation and memory consumption. For an operation decided to use FP16 on TC by performance modeling, the computation overhead (with casing cost) is easily surpassed by the performance benefit: in our evaluation, an operation decided to use FP16 on TC by performance modeling can typically gain about 2x performance improvement (compared with using FP32 on regular CUDA cores), while padding usually leads to less than 20% performance overhead. To consider the performance overhead of zero padding in performance modeling, we can introduce a threshold, which is an empirical estimation on the padding overhead. Only when the casting cost plus this threshold is smaller than performance benefit, we use FP16 on TC.

The memory overhead of padding is usually less than 1%, which is very small. This is because in practice, the number of zero-filled rows or columns via padding is less than 8 and the number of dimensions requiring padding is typically at most

2, while the total number of rows or columns is hundreds.

5 Implementation

Campo extends the mixed precision graph optimizer and runtime system in TensorFlow v1.15. Such an extension includes 235 C++ LOC. The extension is used to decide if an operation should use FP16 based on performance modeling. We add APIs `CheckAllowListOps` and `CheckIfAllowThroughClear` to implement the first and fourth graph traverses. The other two traverses extend the existing implementation for assigning precision to operations in TensorFlow. In TensorFlow's `op_kernel` module, we add an API `InputShapeTranform` to implement input padding and meet the TC requirements on the input shape. Besides the above extension, Campo includes graph profiling and performance modeling (including offline tools to build performance models). In total, Campo is written in 2570 LOC.

6 Discussions

Differences between using performance modeling and static profiling. The static profiling is an alternative approach to get performance of operations in low precision. Using static profiling, the user must collect the information on tensor shapes from operators, and then use the collected information to run operators in low precision offline. We discuss the differences between performance modeling and static profiling as follows.

There are two differences. First, the static profiling has to be done for each NN model and is not scalable, while the performance modeling, once built, can generally work for most NN models. Second, when the number of tensor shapes and operations in an NN model for profiling is small, the static profiling is a better solution to get operation performance in low precision. However, the profiling cost must be small enough to enable practical deployment of static profiling. In contrast, the performance modeling does not incur deployment cost for most of NN models. The performance models can be repeatedly used for NN models, which amortizes the model construction cost.

Portable performance modeling. Our performance modeling is architecture-dependent, because it collects architecture-dependent performance events as the model features. This means that we must build different performance models for different GPU architectures. How to reduce human efforts to build performance models remains to be studied. In addition, it would be interesting to extend Campo to lower precisions (e.g., INT8 and BF16) using the same methodology in Campo. We leave them as our future work.

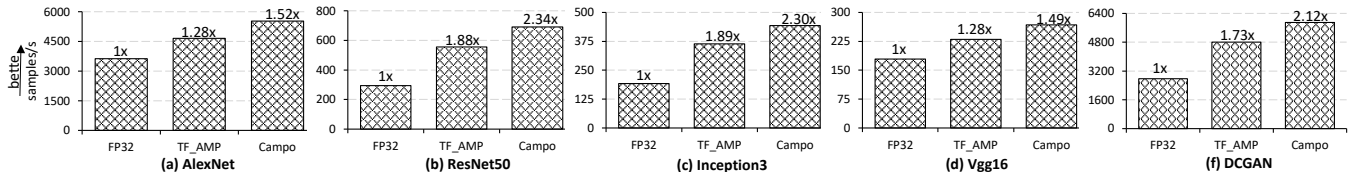


Figure 4: Training throughput with FP32, TF_AMP and Campo on RTX 2080 Ti.

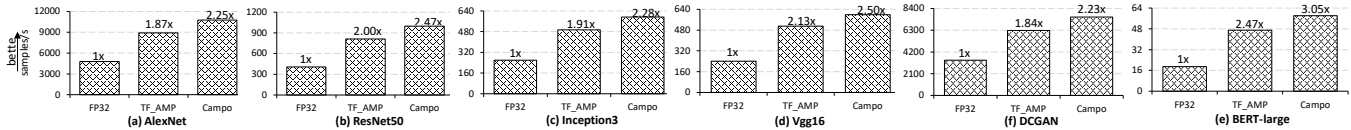


Figure 5: Training throughput with FP32, TF_AMP and Campo on V100.

Table 2: Hardware configurations

CPU	Intel Xeon CPU E5-2648L v4@ 1.80GHz
Main Memory	64 GB DDR4
CPU Cores	2 sockets, 14 cores per socket
GPU	NVIDIA GeForce RTX 2080 Ti (Turing)
CUDA Cores	4352 CUDA cores (68 SMs, 1.54GHz)
Tensor Cores	544 tensor cores
L1 Cache	64 KB (per SM)
L2 Cache	5.767 MB
GPU Device Memory	11 GB GDDR6
GPU	NVIDIA Tesla V100 (Volta)
CUDA Cores	5376 CUDA cores (84 SMs, 1.53GHz)
Tensor Cores	672 tensor cores
L1 Cache	128 KB (per SM)
L2 Cache	6.144 MB
GPU Device Memory	32 GB HBM2

7 Evaluation

7.1 Experimental Setup

Experimental Platforms and Tools. We use a multicore machine equipped with two TC-supported GPUs (i.e., Nvidia GeForce RTX 2080 Ti and V100) and Intel Xeon CPU listed in Table 2. The two GPUs are attached to the server by PCIe 3.0. We use CUDA 9.0 [18], NVIDIA cuDNN 8.0, and Ubuntu 18.04. We use dlprof [14] and Nvidia Nsight Compute [19] to collect performance statistics. We measure system power for GPU, CPU and DRAM by using a collection of industry-standard tools including NVIDIA System Management Interface [20] and Intel Running Average Power Limit (RAPL) Interface [21]. We use the number of samples processed per second and training throughput per Watt as metrics to quantify training throughput and energy efficiency respectively. Unless indicated otherwise, the reported results are collected on V100 and all tests use the default GPU setting.

Benchmarking Methodology. We evaluate six NN mod-

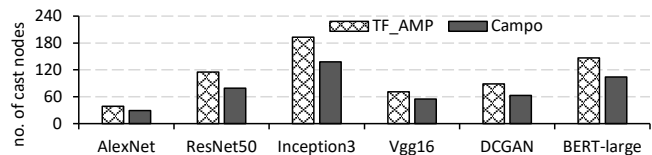


Figure 6: The number of cast nodes of NN models trained with TF_AMP and Campo, respectively.

els including AlexNet [22], Inception3 [23], Vgg16 [24], ResNet50 [25], DCGAN [26], and BERT-large [6]. We only evaluate BERT-large on V100 because of the out-of-memory error on RTX 2080 Ti. For the first four models, Imagenet is used as the training dataset [27]. For DCGAN and BERT-large, we use CelebA [28] and SQuAD [29] as the training dataset, respectively. The training batch size for AlexNet and other models on GeForce RTX 2080 Ti GPU is 256 and 64 respectively, according to the model configurations in related work [30]. The training batch size for BERT-large and other models on V100 is 10 and 256 respectively, according to the model configurations in related work [31].

We run each NN model training experiment ten times and then report the average results. We use TensorFlow v1.15 and its performance optimizer “TF_AMP” for mixed precision training. TF_AMP is our baseline for performance comparison. TF_AMP in TensorFlow v1.15 is the state-of-the-art solution and *the most recent* performance optimizer for mixed precision training. To preserve small gradient values, we adopt the automatic loss-scale optimizer in TF_AMP. To evaluate model accuracy, We test both the Top-1 and Top-5 accuracy on the ImageNet-1k validation set for AlexNet, ResNet50, Inception3 and Vgg16, and the celebA validation set for DCGAN. For BERT, we test the F1 score on the SQuAD v1.1 validation set.

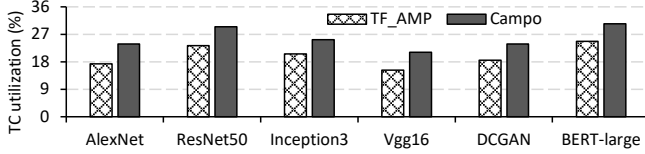


Figure 7: TC utilization of NN models trained with TF_AMP and Campo, respectively.

7.2 Training Throughput

Figures 4 and 5 show the training throughput of Campo, TF_AMP, and single precision training (using FP32). To calculate speedup, the throughput of using the single precision training is used as the baseline.

Using mixed precision training, TF_AMP and Campo achieve large speedup (1.28x - 3.05x) over FP32 on both GPUs. Furthermore, Campo outperforms TF_AMP by 20.8% on average (up to 24.5%) on RTX 2080 Ti, as well as by 20.9% on average (up to 23.4%) on V100. The performance benefit of Campo over TF_AMP comes from two perspectives: reducing the number of cast operation nodes in the dataflow graph and using TC more often, discussed as follows.

7.3 Performance Breakdown

Number of cast operation nodes. Figure 6 quantifies the number of cast operation nodes. Campo reduces the number of cast operations nodes for higher performance. Campo uses 27.7% less cast operation nodes on average (up to 31.3%) than TF_AMP.

TC utilization is defined as the percentage of training time when TC is busy. Figure 7 shows TC utilization. Campo increases the utilization of TC by 29.4% on average (up to 37.9%), which indicates that Campo uses TC more often.

Contribution quantification of the graph rewriting and improving TC utilization. We disable our method of improving TC utilization but keep the graph rewriting to quantify its contribution to the performance improvement (compared with TF_AMP). Then we enable our method of improving TC utilization along with the graph rewriting to quantify the contribution of improving TC utilization. Figure 8(a) and Figure 8(b) show the results on two GPUs. The two figures reveal that the graph rewriting contributes more than improving TC utilization: 84.5% of the overall performance improvement (on average) comes from the graph rewriting.

We also notice that V100 benefits 38.1% more (on average) from TC, compared with RTX 2080 Ti. This is because of two reasons. (1) V100 has more computation resource: V100 has 23.5% more tensor cores than RTX 2080 Ti; (2) training on V100 is able to run 10.4% more operations on TC than training on RTX 2080 Ti, because higher performance benefits of using TC on V100 offset casting cost in more operations.

Table 3: Model accuracy of NN models trained with FP32, TF_AMP and Campo, respectively.

NN models	Top-1 Accuracy (%)			Top-5 Accuracy (%)		
	FP32	TF_AMP	Campo	FP32	TF_AMP	Campo
AlexNet	63.39	64.41	64.38	81.24	81.21	81.19
ResNet50	78.77	78.74	78.75	94.86	94.82	94.85
Inception3	78.42	78.45	78.43	90.15	90.16	90.15
Vgg16	71.58	71.6	71.57	88.28	88.25	88.27
DCGAN	80.12	80.16	80.13	92.47	92.46	92.44
BERT-large	91.35	91.36	91.33	N/A		

7.4 Training Accuracy

Table 3 reports the model accuracy of six NN models trained with FP32, TF_AMP and Campo, respectively. We can see that across NN models, the model training with Campo leads to no loss in model accuracy compared to TF_AMP, which closely matches the FP32 training accuracy (the subtle differences across FP32, TF_AMP and Campo in accuracy are within typical bounds of run-to-run variations).

Campo preserves training accuracy, because of two reasons. (1) Those operations that are numerical unsafe still use FP32. (2) Campo uses the same effective loss-scaling optimizer as TF_AMP to preserve small gradients in FP16.

7.5 Prediction Accuracy of Performance Models

To evaluate the accuracy of the performance models, we use a metric denoted by M_A as follows.

$$M_A = 1 - \frac{1}{n} \sum_{i=1}^n \left| \frac{\hat{y}_i - y_i}{y_i} \right| \quad (3)$$

where n is the number of test cases, and \hat{y}_i and y_i are the predicted and measured execution time for the test case i .

We test 150 performance models for 143 operations respectively. For each model, we use 100 different input sizes as test cases. In total, there are 15000 tests. We report the modeling accuracy for five common operations, i.e., Cast, MatMul, Conv2DBackpropFilter, Conv2DBackpropInput and Conv2D in Figure 9.

In general, the average prediction errors for the five operations are less than 5%, which demonstrates high prediction accuracy. Overall, the prediction error for 143 operations is 5.8% on average (and less than 6%).

Handling mis-prediction of performance modeling. When a mis-prediction happens, there are two possible outcomes. (1) Based on performance modeling, the operation is not scheduled to run in FP16, although it should be for better performance. (2) Based on performance modeling, the operation is scheduled to run in FP16, although it should not, because of high casting cost.

For the case (1), mis-prediction does not cause any performance loss, compared with using full precision (i.e., the

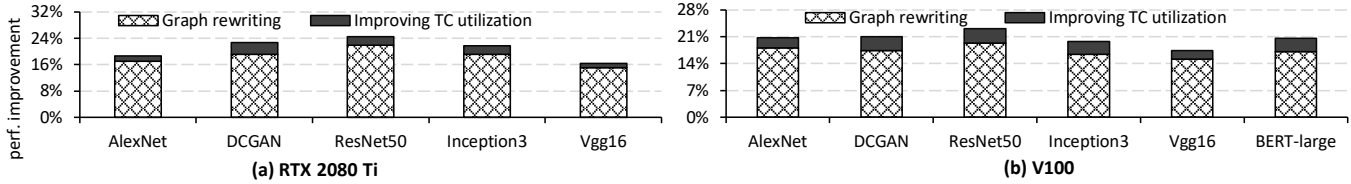


Figure 8: Breakdown of the overall performance improvement from graph rewriting and improving TC utilization.

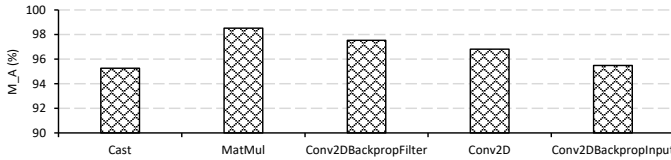


Figure 9: Performance prediction accuracy for five operations based on the operation-specific performance models

Table 4: Average system power consumption of NN models trained with FP32, TF_AMP and Campo on RTX 2080 Ti and V100, respectively.

NN Models	Average System Power (W)					
	RTX 2080 Ti			V100		
	FP32	TF_AMP	Campo	FP32	TF_AMP	Campo
AlexNet	274	268	267	325	319	316
ResNet50	272	265	263	324	313	311
Inception3	273	264	263	326	316	315
Vgg16	273	267	267	324	316	316
DCGAN	275	268	267	327	320	319
BERT-large	N/A	N/A	N/A	332	320	318

original execution). For the case (2), mis-prediction causes performance loss in that operation. But since the performance prediction accuracy is high, the performance loss is easily outweighed by the performance benefit of correctly using FP16 in other operations. In our evaluation, for each NN model, the case (2) happens at most 7 times, taking less than 1.5% of all prediction cases.

No matter whether the case (1) or (2) happens, neither of them causes any loss in training accuracy of the NN model, because the performance modeling is never applied to any numerical-unsafe operation and hence the mis-prediction never happens to any of them.

7.6 Power Consumption and Energy Efficiency

Power consumption. Table 4 summarizes system power consumption. Using TF_AMP and Campo, the system consumes less power than using FP32 across NN models by 6 - 13 Watts, because of the use of power-efficient TC in mixed precision training. Table 4 also shows that using Campo, the system consumes less power than using TF_AMP, due to better utilization of TC in Campo.

Energy efficiency. Figure 10 shows the energy efficiency

of NN models trained with FP32, TF_AMP and Campo. TF_AMP and Campo outperform FP32 by 109.5% and 154.6% on average, respectively, because of the use of reduced precision and power efficient TC. Compared to TF_AMP, Campo outperforms TF_AMP by 21.4% on average (up to 24.2%). This improvement comes from the fact that Campo leads to better performance (see Figure 5) without causing extra power consumption (see Table 4).

8 Related Work

Mixed precision for NN training. Many research efforts have been dedicated to achieve more efficient NN training with mixed precision. Mixed precision training was first introduced by Micikevicius et al [1]. Since then, Nvidia depicts how to use it with TC [3]. Jia et al. [2] use mixed precision to improve scalability of synchronized stochastic gradient descent (SGD) optimizers in NN models without losing model generability. Kuchaiev et al. [32] presents a TensorFlow-based toolkit for mixed precision training of sequence-to-sequence models, with an implementation of a wrapper around the standard TensorFlow performance optimization facility for mixed precision training. Besides the floating-point based mixed precision training, Das et al. [4] are the first to propose mixed precision training of convolutional neural networks using integer operations on ImageNet-1K dataset. Svyatkovskiy et al. [33] introduce a learning rate schedule for training distributed deep recurrent neural networks with mixed precision on GPU clusters. Their schedule facilitates neural network convergence at up to O(100) workers.

Different from the above efforts, our work reveals the overlooked performance issues related to casting cost in mixed precision graph optimization.

Mixed precision for other GPU applications. Mixed precision has been explored to speed up dense linear system solvers [34,35] and WZ factorization [36] in the context of HPC. Kotipalli et al. [37] present AMPT-GA, an automatic mixed precision optimization system that automatically selects the optimal data precision to maximize performance while meeting accuracy constraints for GPU Applications. However, its evaluation is only limited to an NVIDIA Tesla P100 GPU machine without TC. Haidar et al. [38] apply mixed-precision FP16-FP32/FP64 to a high-performance iterative refinement solvers and take advantage of TC. Gallo et al. [39] propose the concept of mixed-precision in-memory

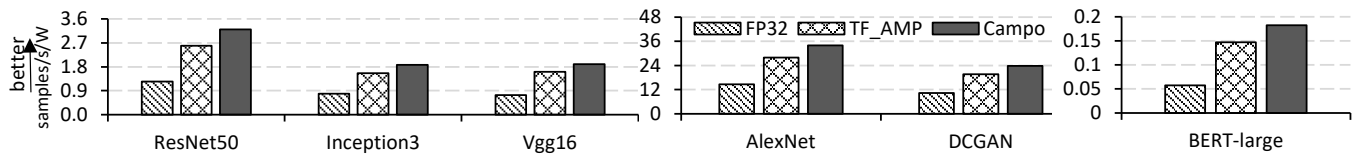


Figure 10: Energy efficiency of NN models trained with FP32, TF_AMP and Campo.

computing with a combination of a von Neumann machine and a computational memory unit. Baboulin et al. [40] leverage mixed precision to accelerate computations in many dense and sparse linear algebra algorithms. Lam et al. [41] introduce a framework that employs binary instrumentation and modification to build mixed-precision configurations of existing binaries originally developed for the use of double precision. In contrast, our work focuses on applying mixed precision to NN training.

Investigation on the usage of TC. Nvidia TC follows the IEEE 754 standard [42] and utilizes mixed precision with matrix multiplication input in FP16 and accumulation in FP32. Motivated by the benefits of TC, Yan et al. [43] demystify how TC on Turing GPUs works and implement TC-based HGEMM on NVIDIA Turing GPUs. Markidis et al. [44] perform a performance evaluation of TC in mixed precision as well as three different approaches of programming matrix-multiply-and-accumulate on TC on V100. Brennan et al. [45] perform a thorough analysis of the effects of low precision operations and TC on graph convolutional neural networks. Abdelfattah et al. [46] explore leveraging TC to implement an optimized batched Matrix multiplication (HGEMM) in half-precision arithmetic. Our work is different from these efforts, as we explore the use of TC in NN training.

9 Conclusions

This paper introduces Campo, a cost-aware performance optimization tool for mixed-precision NN training that assigns the optimal precision (either FP32 or FP16) to training operations while minimizing the unnecessary casts to maximize the training performance. Campo is based on our unique observation that the casting cost to achieve mixed precision training may offset the performance benefit of using low precision in mixed precision training. This observation is ignored in the existing approaches of using mixed precision, which leads to smaller performance improvement or even performance loss. We build operation-specific performance models to predict and quantify the impact of casting cost on the performance of using low precision. With the performance models, at runtime Campo employs a cost-aware graph rewriting strategy to make decisions on which precision should be used for each operation without losing NN training accuracy. We evaluate Campo with six NN models on Nvidia Turing and Volta architecture-based GPUs, and show that Campo largely outperforms TensorFlow.

10 Acknowledgements

We thank anonymous reviewers and our shepherd for their valuable feedback. This work is partially supported by the National Science Foundation of China under grants 61972137 and 61772183.

References

- [1] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.
- [2] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.
- [3] Nvidia. Nvidia’s mixed-precision training - tensorflow example. <https://docs.nvidia.com/deeplearning/performance/mixed-precision-training>, 2018.
- [4] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, et al. Mixed precision training of convolutional neural networks using integer operations. *arXiv preprint arXiv:1802.00930*, 2018.
- [5] Google. Tensorflow - enable mixed precision graph rewrite. https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/train/experimental/enable_mixed_precision_graph_rewrite, 2018.
- [6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [7] Muthian Sivathanu, Tapan Chugh, Sanjay S Singapuram, and Lidong Zhou. Astra: Exploiting predictability to

- optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 909–923, 2019.
- [8] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 875–890, 2020.
- [9] Xufan Zhang, Ziyue Yin, Yang Feng, Qingkai Shi, Jia Liu, and Zhenyu Chen. Neuralvis: Visualizing and interpreting deep learning models. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1106–1109. IEEE, 2019.
- [10] Jiawen Liu, Dong Li, Gokcen Kestor, and Jeffrey Vetter. Runtime concurrency control and operation scheduling for high performance neural network training. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 188–199. IEEE, 2019.
- [11] Xin He, Jiawen Liu, Zhen Xie, Hao Chen, Guoyang Chen, Weifeng Zhang, and Dong Li. Enabling energy-efficient dnn training on hybrid gpu-fpga accelerators. In *Proceedings of the ACM International Conference on Supercomputing*, pages 227–241, 2021.
- [12] Jiawen Liu, Hengyu Zhao, Matheus A Ogleari, Dong Li, and Jishen Zhao. Processing-in-memory for energy-efficient neural network training: A heterogeneous approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 655–668. IEEE, 2018.
- [13] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 891–905, 2020.
- [14] Dlprof - nvidia deep learning frameworks documentation. <https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide/>, 2021.
- [15] Marco Tagliasacchi, Beat Gfeller, Félix de Chaumont Quiry, and Dominik Roblek. Self-supervised audio representation learning for mobile devices. *arXiv preprint arXiv:1905.11796*, 2019.
- [16] Thomas W MacFarland and Jan M Yates. Spearman’s rank-difference coefficient of correlation. In *Introduction to nonparametric statistics for the biological sciences using R*, pages 249–297. Springer, 2016.
- [17] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. Mlperf training benchmark. *arXiv preprint arXiv:1910.01500*, 2019.
- [18] Cuda toolkit documentation v9.0. <https://developer.nvidia.com/cuda-toolkit-archive>, 2019.
- [19] Nvidia nsight compute - nvidia developer documentation. <https://developer.nvidia.com/nsight-compute>, 2021.
- [20] Nvidia. Nvidia system management interface. <https://developer.nvidia.com/nvidia-system-management-interface>, 2018.
- [21] Intel’s runningaverage power limit (rapl) interface. <https://01.org/rapl-power-meter>, 2019.
- [22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [23] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.
- [24] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [26] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.
- [27] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. IEEE, 2009.

- [28] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.
- [29] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics.
- [30] Chuan Li. Rtx 2080 ti deep learning benchmarks with tensorflow. <https://lambdalabs.com/blog/2080-ti-deep-learning-benchmarks/>, 2019.
- [31] Nvidia data center deep learning product performance. <https://developer.nvidia.com/deep-learning-performance-training-inference>, 2020.
- [32] Oleksii Kuchaiev, Boris Ginsburg, Igor Gitman, Vitaly Lavrukhin, Jason Li, Huyen Nguyen, Carl Case, and Paulius Micikevicius. Mixed-precision training for nlp and speech recognition with openseq2seq. *arXiv preprint arXiv:1805.10387*, 2018.
- [33] Alexey Svyatkovskiy, Julian Kates-Harbeck, and William Tang. Training distributed deep recurrent neural networks with mixed precision on gpu clusters. In *Proceedings of the Machine Learning on HPC Environments*, pages 1–8. 2017.
- [34] Alfredo Buttari, Jack Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, and Jakub Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *The International Journal of High Performance Computing Applications*, 21(4):457–466, 2007.
- [35] Azzam Haidar, Panruo Wu, Stanimire Tomov, and Jack Dongarra. Investigating half precision arithmetic to accelerate dense linear system solvers. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 1–8, 2017.
- [36] Beata Bylina and Jarosław Bylina. Mixed precision iterative refinement techniques for the wz factorization. In *2013 Federated Conference on Computer Science and Information Systems*, pages 425–431. IEEE, 2013.
- [37] Pradeep V Kotipalli, Ranvijay Singh, Paul Wood, Ignacio Laguna, and Saurabh Bagchi. Ampt-ga: automatic mixed precision floating point tuning for gpu applications. In *Proceedings of the ACM International Conference on Supercomputing*, pages 160–170, 2019.
- [38] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 603–613. IEEE, 2018.
- [39] Manuel Le Gallo, Abu Sebastian, Roland Mathis, Matteo Manica, Heiner Giefers, Tomas Tuma, Costas Bekas, Alessandro Curioni, and Evangelos Eleftheriou. Mixed-precision in-memory computing. *Nature Electronics*, 1(4):246–253, 2018.
- [40] Marc Baboulin, Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526–2533, 2009.
- [41] Michael O Lam, Jeffrey K Hollingsworth, Bronis R de Supinski, and Matthew P LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 369–378, 2013.
- [42] Nathan Whitehead and Alex Fit-Florea. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus. *rn (A + B)*, 21(1):18749–19424, 2011.
- [43] Da Yan, Wei Wang, and Xiaowen Chu. Demystifying tensor cores to optimize half-precision matrix multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 634–643. IEEE, 2020.
- [44] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE, 2018.
- [45] John Brennan, Stephen Bonner, Amir Atapour-Abarghouei, Philip T Jackson, Boguslaw Obara, and Andrew Stephen McGough. Not half bad: Exploring half-precision in graph convolutional neural networks. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 2725–2734. IEEE, 2020.
- [46] Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. Fast batched matrix multiplication for small sizes using half-precision arithmetic on gpus. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 111–122. IEEE, 2019.



PRIMO: Practical Learning-Augmented Systems with Interpretable Models

Qinghao Hu^{1,2} Harsha Nori³ Peng Sun⁴ Yonggang Wen¹ Tianwei Zhang¹

¹Nanyang Technological University ²S-Lab, NTU ³Microsoft ⁴SenseTime Research

Abstract

While machine learning has demonstrated remarkable performance in various computer systems, some substantial flaws can prohibit its deployment in practice, including opaque decision processes, poor generalization and robustness, as well as exorbitant training and inference overhead. Motivated by these deficiencies, we introduce PRIMO, a unified framework for developers to design practical learning-augmented systems. Specifically, (1) PRIMO provides two interpretable models (PrAM and PrDT), as well as a *Distill Engine*, to support different system scenarios and deployment requirements. (2) It adopts *Bayes Optimization* to automatically identify the optimal model pruning strategy and hyperparameter configuration. (3) It also implements two tools, *Monotonic Constraint* and *Counterfactual Explanation*, to achieve transparent debugging and guided model adjustment. PRIMO can be applied to different types of learning-augmented systems. Evaluations on three state-of-the-art systems show that PRIMO can provide clear model interpretations, better system performance, and lower deployment costs.

1 Introduction

Over the years, machine learning (ML) has been widely adopted to optimize systems across many fields, e.g., storage [29, 82, 85], network [66, 77, 95], security [24, 28, 74], compiler optimization [8, 93, 94] and cluster scheduling [65, 89, 92]. These learning-augmented systems demonstrate marvelous performance compared with conventional heuristic or mathematical optimized systems.

However, most of these applied models are very complex and treated as black-boxes to developers, which brings significant gaps in deploying them in practice. **First, building a production-level learning-augmented system can incur huge costs.** From the experience at Microsoft [42], the model training process could take days to weeks with massive data. Some systems require frequent model updates to adapt to dynamic environment changes, whose cost often exceeds enterprise expectations. For some scenarios with limited data samples, developers have to use techniques to synthesize training samples [12, 45, 96], which inevitably introduce bias to the model and cause performance deterioration in practice [8, 10, 66]. Moreover, the inference process of these complicated models can pose heavy computational pressure to

systems which have high real-time requirements [43, 81, 82], which can significantly restrict parallel capabilities and affect scalability in practice.

Second, the prediction process of these black-box models are unintelligible to humans. Developers lack understanding and trust of the model's behavior [19, 53, 91], which makes it difficult for them to perform model adjustments and ad hoc debugging in practical scenarios. Some efforts have been made to improve system transparency through interpreting black-box models [26, 27, 55]. They typically build *surrogate* models to obtain explanations for individual predictions, thus validating model behaviors and diagnosing system mistakes. However, they cannot provide an interpretation fidelity guarantee, and therefore the corresponding explanations are unreliable and potentially misleading [58, 70]. In addition, they cannot address the aforementioned system cost issue.

In this paper, we aim to resolve the above challenges and facilitate *transparent, accurate* and *lightweight* system deployment in practice. We introduce PRIMO (**P**rior-based **I**nterpretable **M**odel **O**ptimization), the *first* unified framework that assists developers to design and optimize learning-augmented systems with interpretable models. The design of PRIMO is based on two key insights. First, *simple interpretable models have the capability of handling complex system problems*. Interpretable models do not sacrifice prediction accuracy [35, 62, 72], and simple model structures with low resource overhead are very suitable for real-time systems. Their effectiveness is often underestimated [70]. Second, *prior experience and domain knowledge can be leveraged by developers to further optimize the interpretable models* [20, 76], which is hard to achieve for black-box models.

PRIMO makes several innovations to enhance learning-augmented systems. First, to provide comprehensive support for different systems, PRIMO introduces two interpretable model algorithms: PrAM is designed for better prediction accuracy and PrDT applies to systems with strict latency or computation constraints. PRIMO can help developers select a suitable model *automatically* based on their system requirements, including latency, accuracy, and resource budget. In addition to training models directly, PRIMO also supports distilling existing complex models, which applies to exploration-based systems with reinforcement learning (RL) [23, 49, 56, 59].

Second, to fully exploit the potential of interpretable mod-

els, we design several built-in mechanisms to optimize model performance leveraging *prior* information. (1) PRIMO implements *Bayes Optimization* to find the optimal model pruning strategy and hyperparameter configurations for higher prediction accuracy and lower computation overhead. It fully takes advantage of prior search information to minimize the search space and training cost. (2) PRIMO also facilitates model post-processing for developers with their domain knowledge. Specifically, it provides two tools for model adjustment through adding *Monotonic Constraints* and transparent debugging with *Counterfactual Explanations*.

Based on these innovations, PRIMO provides not only precise and comprehensive interpretations for developers to understand and adjust models, but also better prediction accuracy and smaller overhead. To extensively evaluate these benefits in real scenarios, we apply PRIMO to three state-of-the-art learning-augmented systems, including two *online* systems (LinnOS for flash storage [29] and Pensieve for video steaming [51]), and an *offline* system (Clara for SmartNIC offloading [66]). For LinnOS, PRIMO provides a 2.8× system performance improvement, and reduces model training time by over 100×, as well as inference latency by over 20×. For Clara, PRIMO beats a series of black-box models in prediction accuracy and saves over 10× training cost. For Pensieve, PRIMO achieves better generalization ability and a 79× inference latency reduction. We believe PRIMO can bring similar benefits to other learning-augmented systems as well.

To summarize, we make the following contributions:

- To the best of our knowledge, PRIMO is the first framework to provide inherent interpretability for learning-augmented systems development.
- We design built-in mechanisms and adjustment tools for developers to achieve *transparent, accurate* and *lightweight* system deployment in practice.
- For the first time, we demonstrate that simple interpretable models can outperform complex black-box techniques in various real systems.

2 Background and Motivation

2.1 Learning-Augmented Systems

Learning-augmented systems apply machine learning techniques to optimize system performance [42]. They typically build various ML models to obtain preeminent system policies from historical execution data, such as Support Vector Machines (SVM) [65, 66], Random Forest (RF) [5, 86], Gradient Boosting Decision Tree (GBDT) [32, 92]. With the popularity of deep learning (DL) algorithms, they were also introduced to further enhance systems, e.g., Deep Neural Network (DNN) [29, 82], Convolutional Neural Network (CNN) [43, 53], Recurrent Neural Network (RNN) [66, 90] and Reinforcement Learning (RL) [41, 51]. We classify them into the following categories.

Taxonomy. Learning-augmented systems typically follow similar design workflows to integrate ML models into system operations. Based on the optimization type, they can be classified into two categories. (1) **Prediction-based** systems utilize the *supervised learning* paradigm (e.g., classification, regression) to optimize system problems. (2) **Exploration-based** systems usually adopt *reinforcement learning* to learn optimal policies in an explore-exploit way. Since there are relatively fewer unsupervised learning-based systems in practice, we consider them as our future work (§8).

Based on system requirements and application scenarios, learning-augmented systems can be divided into the following two types. (1) **Online** systems require the ML model to make prompt predictions for real-time data. Developers need to consider model inference latency and computation overhead, in addition to prediction accuracy. (2) **Offline** systems usually do not need to deploy ML models for real-time serving and have no latency or computation requirements. These systems are performance-critical and the objective of ML models is to improve prediction accuracy.

PRIMO is designed as a unified framework, providing respective optimization mechanisms for different types of learning-augmented systems.

2.2 Challenges and Motivation

While plenty of work has demonstrated the potential of ML techniques in improving system performance, there exist several challenges in the development and deployment of learning-augmented systems in practice.

Model development. First, building a qualified ML model for the target system has the following two challenges:

- **C1: high training and tuning cost.** As stated by Microsoft AutoSys [42], costs of ML model training often exceeds enterprise expectations. Real system environments are dynamically changing and stale models will cause performance deterioration. Therefore, frequent model fine-tuning or retraining is necessary, which could take days to weeks [42, 52] in order to outperform heuristic algorithms. If there are not enough GPU resources, the update time will become even more intolerable for DL models.
- **C2: susceptible to the quantity and quality of data.** A large amount of high-quality training data are essential to produce satisfactory ML models. However, in some cases, insufficient data [8, 10, 66] or excessive data collection cost [52, 88] hinder developers from training qualified models. Possible solutions include data augmentation and synthesis [12, 45, 96]. Nevertheless, owing to the sophisticated distribution of real-world data, the generated data inevitably introduce bias and shift to the learning model [66], which could compromise the system performance in practice.

Model deployment. Second, deploying ML models in practice has interpretability and inference overhead issues:

- **C3: opaque decision making process.** Developers mainly

Strategies	Interpretation Fidelity	Local Interpretation	Global Interpretation	Transparent Adjustment	Deployment Cost	Accuracy ↗	Roustness ↗	Latency ↘
Black-box models (e.g., <i>DNN, RL, GBDT</i>)	✗	✗	✗	✗	\$\$\$	★	★	★
Interpreting black-box models [26, 67]	✗	✓	✗	✗	\$\$\$	★	★	★
Building interpretable models (PRIMO)	✓	✓	✓	✓	\$	★☆	★☆	★☆

Table 1: Comparisons of different strategies for learning-augmented systems (☆: Performance improvement).

focus on improving key system metrics (e.g., I/O latency [29], user experience [51]) when designing and evaluating ML models, while ignoring their *interpretability*. As a result, most of these learning models are *black-boxes* whose prediction processes are unintelligible to humans [26, 40, 70]. Due to such opacity, system operators cannot guarantee model predictions are risk-free and have insufficient confidence to deploy them.

- **C4: difficulty in troubleshooting and adjustment.** In order to achieve expected performance in production environments, system operators typically need to adjust the learning models according to the actual scenarios [19, 53, 55], including input features alteration, model structure modification, data augmentation, etc. All these actions require the operators to have a profound understanding of the system and the corresponding ML technique [26, 42], which is difficult when the model is complex. In addition, ad hoc debugging is another substantial challenge to learning-augmented systems for black-box models. Improper modifications may cause severe performance degradation.
- **C5: exorbitant deployment overhead.** The model deployment overhead is another key factor for system operators' consideration [53]. The latency and computation requirements of some systems [43, 81, 82] are far more strict than conventional AI tasks. High inference overhead can cause side effects to production workloads and limit their parallelism capability [29], which can further restrict deployment scalability.

2.3 Model Interpretation as a Solution

One possible solution to address the above challenges is model interpretation. There are two primary directions to apply model interpretation for learning-augmented systems.

2.3.1 Interpreting Black-box Models.

The essential idea is to leverage existing interpretation methodologies to interpret the black-box models, making them more intelligible and transparent. A variety of interpretation tools (e.g., Lime [67], Captum [39], Shap [48]) were designed to explain the mechanisms of DNN models for CV and NLP tasks. Similar studies were also performed for other domains. For instance, Lemna [26] employs a mixture regression model [36] to interpret RNN models in DL-based security applications. Metis [55] proposes to interpret networking systems with the decision tree or hypergraph. However, we argue that the idea of interpreting black-box models is not sufficient

for learning-augmented systems for the following reasons.

(1) **No fidelity guarantee.** These tools typically interpret black-box models in a *post hoc* way, where another *local surrogate* model is created to explain the original model. They cannot have a fidelity guarantee with respect to the original model. Therefore, the corresponding explanations are often unreliable, and can be misleading [58, 70]. The fidelity of some widely applied interpretation methods (e.g., attention-based explain [84]) are still in dispute [34, 83]. Appendix B.1 presents an example of contradictory XGBoost explanations.

(2) **Limited interpretation.** Most existing tools (e.g., Lime, Lemna) focus on explaining individual predictions (local interpretation) instead of the entire model behavior (global interpretation). Thus, the interpretation results typically cannot yield enough information for system troubleshooting. Appendix B.2 shows their insufficiency for global understanding and model surrogate.

(3) **Requiring domain knowledge.** Different systems may employ different models and algorithms. There is no unified tool that can provide comprehensive support for interpreting arbitrary models. Consequently, domain knowledge and manual efforts are required to implement the tools and understand the explanation results. This poses a huge challenge for developers to design a learning-augmented system.

(4) **Incapability of handling other challenges.** Those tools only focus on model interpretation and understanding (C3 & C4), but ignore other challenges discussed in §2.2.

2.3.2 Building Interpretable Models

A more promising direction, which is adopted in PRIMO, is to train *interpretable models* directly for learning-augmented systems. Interpretable models refer to the models that are inherently intelligible, where their explanations provided by themselves are faithful to what the models actually compute [58, 70]. Common interpretable models include linear regression, logistic regression, decision tree, decision list, etc. They have great potential to enhance different types of learning-augmented systems.

According to our observation from recent state-of-the-art learning-augmented systems [1], the scale of models in these systems tend to be relatively smaller than popular production-level AI models (although they are still too complex for humans to understand). For instance, the number of neurons in a RL-based system is typically less than 10K [19]. This is because most data samples in learning-augmented systems are well structured, with good representations in terms of naturally meaningful features. In such scenarios, a much sim-

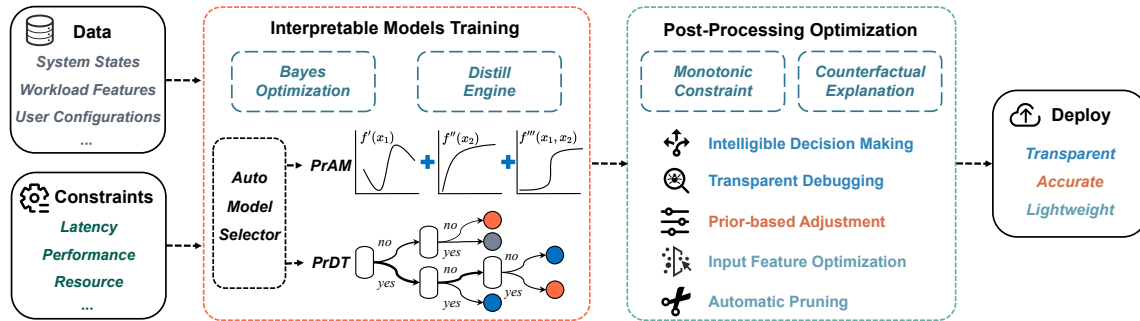


Figure 1: The workflow of learning-augmented system development using PRIMO.

pler interpretable model can give comparable performance to complex black-box models [70]. Therefore, developers can employ interpretable models for their systems, which require less data, training and tuning cost (C1 & C2). The models give more information for system operators to understand (C3), troubleshoot and adjust (C4), and the inference speed is much faster than the original black-box model (C5).

Summary. The benefits of PRIMO compared with other methods are summarized in Table 1. It can provide not only highly precise and comprehensive interpretations for developers to understand and adjust models, but also higher accuracy and robustness, and smaller training and inference overhead. These greatly facilitate model deployment in practice.

3 PRIMO Design

We introduce PRIMO, a unified framework that assists developers to design practical learning-augmented systems. Particularly, (1) we employ *transparent* and *deterministic* interpretable models to circumvent the uncertainty issues of black-box model inference. (2) We integrate new tools for developers to leverage prior knowledge to optimize interpretable models *automatically*. (3) We design a built-in mechanism to search optimal hyperparameters in a *fast* and *convenient* way, without extra effort from the developers. Based on these designs, PRIMO can address all the challenges in §2.2.

3.1 Framework Overview

PRIMO optimizes both the *training* and *post-processing* stages of building learning-augmented systems. Figure 1 illustrates the development workflow with PRIMO. In the model training stage, PRIMO provides two interpretable model algorithms (PrAM and PrDT) designed for different system scenarios¹. PRIMO helps developers automatically select suitable algorithms based on their system requirements including latency, accuracy, and resource budget. It supports training the interpretable model directly, or converting an existing complex black-box model into a simple interpretable model through the *Distill Engine*. We also leverage *Bayes Optimization* to find the optimal model pruning strategy and hyperparameter configurations for higher prediction accuracy and lower

¹Other interpretable models can also be conveniently integrated into this framework, which will be considered in our future work.

computation overhead. After the model is trained, PRIMO offers several optimization tools in the post-processing stage, e.g., prior-based model adjustment through adding *monotonic constraints*, transparent debugging with *counterfactual explanations*. Below we detail the mechanism of each component.

3.2 Interpretable Models

As introduced in §2.1, different system scenarios have different requirements for the learning models. To this end, PRIMO employs two types of interpretable model algorithms: PrAM is designed for better prediction accuracy and PrDT applies to systems with strict latency constraint or computation sensitivity. PRIMO supports automatic model selection based on the demands specified by the developers.

3.2.1 PrAM: Addictive Model based Method

Our first interpretable model, PrAM, is based on the Standard Generalized Additive Models (GAMs) [30]. GAMs consist of a series of *shape functions* $f_i(\cdot)$ and an intercept μ_0 (Equation 1). Since each shape function considers only one univariate term (the i th feature x^i) and their combination is additive, GAMs are interpretable: we can clearly understand the contribution of each single feature to the final prediction.

Compared with linear interpretable models (e.g., logistic regression), GAMs can cope with more complex prediction tasks because shape functions are typically nonlinear and have better fitting capability. To further increase model performance, we adopt the state-of-the-art GAM algorithm: GA²M [47], which additionally considers the interactions of two features and maintains the interpretability (more details are in Appendix A.1). GA²M has the following form:

$$g(E[y | \mathbf{x}]) = \underbrace{\mu_0 + \sum f_i(x^i)}_{\text{GAM}} + \underbrace{\sum f_{ij}(x^i, x^j)}_{\text{Interactions}} \quad (1)$$

where $g(\cdot)$ is a link function that adapts GA²M to different tasks, e.g., regression (identity), classification (logistic function); $f_{ij}(\cdot)$ represents the interaction effect of features i and j , which can be visualized as a two-dimensional heatmap.

In our implementation, PrAM extends the open-source library EBM [63] to obtain the optimal model with high compactness and accuracy. Compared to the complex DL models, PrAM can not only provide interpretability, but also takes less

training resources (without the need of GPUs) and training data samples, significantly reducing the training time and cost.

3.2.2 PrDT: Decision Tree based Method

Our second interpretable model PrDT is constructed from Decision Trees (DTs). DTs are binary tree-structured models where each *branch node* tests a condition and each *leaf node* makes a prediction [71]. Because DTs are non-parametric and can be essentially expressed as an equivalent rule list, they are transparent and simple to interpret how a prediction is obtained. Besides, the decision-making processes of DTs can be visualized so developers can easily adjust the trees according to the system requirements. They present powerful prediction capability for both classification and regression tasks, even compared with complex black-box models.

In addition to the excellent interpretability and accuracy, DTs have extremely low computation overhead and inference latency. Consequently, they are applicable to many scenarios with strict latency and resource constraints [29, 61]. Besides, DTs also exhibit other benefits, including robust performance under dynamic system environments, requiring less training data and no data preprocessing overhead during inference.

It is necessary to optimize the complexity of a DT to avoid the overfitting issue, which can affect the model generalization, accuracy and computation overhead. Instead of adding constraints (e.g., maximum depth, minimum number of samples for a leaf node) during DT training, PrDT trains a full decision tree without any limitation to capture more information from the training dataset. We adopt *minimal cost-complexity pruning* [14] to prune the full tree in the *post-processing* stage, which is elaborated in Appendix A.2.

3.3 Model Training

PRIMO supports two training modes. (1) *Direct*: the developer can train an interpretable model from scratch. This applies for most **prediction-based** systems. (2) *Distill*: the developer can generate an interpretable model from the original black-box model through the *Distill Engine*. This is mainly for **exploration-based** systems. To obtain high-quality models, both modes support the integration of *Bayes Optimization* for efficient model structure and hyperparameter search.

3.3.1 Bayes Optimization

There exists a trade-off between the model complexity and accuracy for both interpretable models. In order to find accurate and succinct models, PRIMO leverages Bayes Optimization (BO) [76], an iterative algorithm to automatically search for the optimal model configurations.

Objective function. For both PrAM and PrDT, we build a universal model scoring function $S(\theta)$ to quantify the model performance and complexity as the search objective:

$$S(\theta) = P(\theta) + \lambda \cdot C(\theta)^\gamma \quad (2)$$

where $P(\theta)$ represents the model performance (e.g., classification accuracy) under hyperparameters θ during validation;

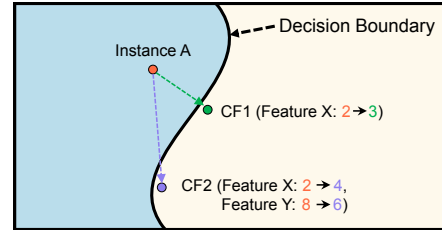


Figure 2: Illustration for the counterfactual explanation.

λ is a knob that controls the model complexity according to users' preference; $C(\theta)$ is a metric for model complexity. For PrDT, $C(\theta^{\text{PrDT}}) = N_{\text{leaves}} \times N_{\text{depth}}$, where we consider both the number of tree leaves and tree depth since unbalanced-deeper trees typically cost longer condition inference time. For PrAM, $C(\theta^{\text{PrAM}}) = N_{\text{interactions}} \times N_{\text{maxbins}}$, where both the number of feature interaction terms and maximum number of bins in the feature histogram are included. Besides, the normalization factor γ regulates the effect of the model complexity.

Prior-based hyperparameter search. Specifically, PRIMO employs Gaussian Process (GP) as the probabilistic surrogate model of the objective function $S(\theta)$ in Equation 2. The prediction of GP follows a normal distribution: $p(S | \theta, \Theta) = \mathcal{N}(S | \hat{\mu}, \hat{\sigma}^2)$, where Θ indicates the hyperparameter search space. To determine which point should be evaluated next, PRIMO adopts expected improvement (EI) as the acquisition function to trade-off exploration and exploitation [20]. In each iteration, PRIMO generates a set of hyperparameters and evaluates them on the interpretable model to obtain new results which are used to update the surrogate model. Compared with Grid Search (GS) and Random Search (RS) [13], BO is more efficient since it fully utilizes the prior information to minimize the search space. For instance, as shown in Figure 17 in Appendix A.2, BO can rapidly reduce the search space to a smaller size ($10^{-5} \sim 10^{-2}$) for a better focus.

3.3.2 Distill Engine

In some scenarios, the learning models require special optimization. For instance, LinnOS [29] leverages *biased training* to reduce the false submit rate while causing the higher false revoke rate. PRIMO introduces the *Distill Engine*, which can build an interpretable surrogate model to approximate the behavior of the original black-box learning model using knowledge distillation [7, 31].

Another application of the *Distill Engine* is RL policy extraction. Both PrAM and PrDT work well for **prediction-based** systems using supervised learning, but are less supportive for **exploration-based** systems due to their incompatibility with RL. A series of works [11, 69, 75] have demonstrated the feasibility of converting NN-based learning policies to an interpretable models. PRIMO adopts Viper [11] to perform RL policy extraction. Specifically, we collect the trajectories of $\{\mathbf{s}_i, a_i\}$ pairs (i.e., system states \mathbf{s}_i and actions a_i of learned policy $\pi(\mathbf{s}_i, a_i)$) generated by the original RL model and perform supervised learning to build the interpretable models.

	System Scenario	ML Algorithm	Type	Primo
LinnOS [29]	Flash Storage I/O	DNN	Online	PrDT (Direct)
Clara [66]	SmartNIC Offloading	Mixture (LSTM, GBDT, SVM)	Offline	PrAM (Direct)
Pensieve [51]	Video Streaming	RL	Online	PrDT (Distill)

Table 2: Summary of case studies for PRIMO evaluation.

To obtain a robust policy, we augment the poor-performing pairs and train the model iteratively until it is converged.

3.4 Post-Processing Optimization

After the interpretable model is built, developers can use their prior knowledge to further optimize the model and enhance the system performance. PRIMO designs two tools to assist developers in model post-processing. Note that these operations are *optional* since generally the trained interpretable models already achieve satisfactory performance.

3.4.1 Monotonic Constraint

In many learning-augmented systems, the input features exhibit a monotonic relationship with the output values (e.g., higher video bitrate selection with better bandwidth). But the corresponding model often presents a non-monotonic pattern due to the sub-optimal construction strategy or noisy training data (e.g., outlier data points, biased synthetic data). This can lead to unstable performance and intelligibility degradation in practice. To this end, PRIMO leverages a method from DP-EBM [62], which adds monotonic constraints to boosted trees via post-processing. Specifically, we model this task as an isotonic regression problem [15] with respect to a complete order. The objective is to minimize $\sum_i w_i (y_i - \hat{y}_i)^2$ subject to $\hat{y}_i \leq \hat{y}_j$ and weights w_i are strictly positive. We adopt the Pool Adjacent Violators (PAV) [6] algorithm to obtain an optimal solution maintaining monotonicity, and use it to replace the original shape function of PrAM. Our tool only needs developers to provide the feature name or index and the subsequent model adjustment process is transparent and automatic.

3.4.2 Counterfactual Explanation

To make modifications to the models, developers need to answer some challenging questions, e.g., which feature related shape function should be adjusted? how to determine the modification degree? To help them make reasonable decisions, we design the Counterfactual Explanation tool in PRIMO to generate additional insights for model adjustment. As illustrated in Figure 2, this tool aims to find smaller change (green arrow) to the feature values that can alter the prediction to a predefined output within the dataset. It typically uses the k -nearest neighbors (kNN) algorithm to find k training instances with the minimum L_2 distances [80]. To address the inefficiency of the brute-force kNN approach, we propose to use Ball Tree [22] to partition data in a series of nesting hyper-spheres, thus the distance between a prediction point and the centroid is sufficient to determine a lower and upper bound on the distance to all points within the hyper-sphere node. This

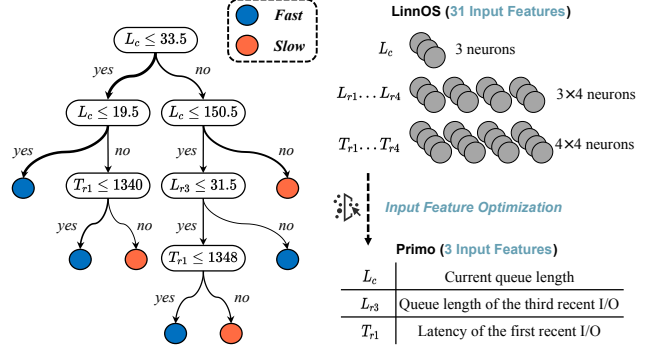


Figure 3: (Left) Learned PrDT model for an SSD. The thicker arrow line denotes the higher frequency. (Right) PRIMO optimizes the input features of LinnOS. Each feature represents a *digit* in LinnOS while a complete *number* in PRIMO.

approach considerably reduces the query time when dealing with large-scale and high-dimensional datasets. And developers could perform guided model adjustment easily.

PRIMO Experiments. In the following three sections, we will present three case studies to demonstrate how PRIMO can optimize state-of-the-art learning-augmented systems. Table 2 describes these three scenarios. The key observation for each case is summarized in Appendix C. We believe PRIMO can be applied to other learning-augmented systems as well.

4 Case Study 1: LinnOS

As the first case, we consider LinnOS [29], a learning-based operating system that accelerates storage applications. LinnOS adopts a 3-layer neural network (31-256-2, in total of 8706 parameters) for *each* SSD to precisely predict its performance. To achieve this, it collects the traces of real workloads running on the SSD and obtains fine-grained information (per I/O), including recent queue lengths and latency. Instead of predicting the concrete latency values, LinnOS simplifies it as a binary (fast / slow) classification task through setting an inflection point (*IP*). More details about LinnOS and our implementation can be found in Appendix D.1.

PRIMO automatically selects the PrDT model for LinnOS, since it has comparable accuracy and lower inference latency than PrAM. For comprehensive evaluation, we consider two models with different optimization objectives: *efficiency*-oriented (PRIMO-E) and *performance*-oriented (PRIMO-P). We compare PRIMO with two baselines. (1) Base: the vanilla Linux I/O mechanism. (2) LinnOS: we set the inflection point of LinnOS as a constant percentile (at p85 latency) and apply the biased loss to the model training (all keep the same).

4.1 System Interpretation

The primary goal of PRIMO is to provide interpretation for the target system. Figure 3 (left) presents the learned decision tree (PRIMO-E) for one SSD. The explanation of each notation can be found on the right side. From this tree, we can clearly understand how PRIMO makes decisions for each prediction

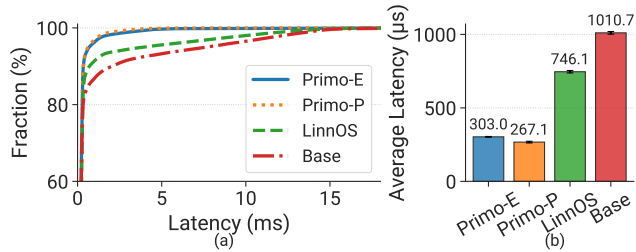


Figure 4: Overall performance comparisons. (a) CDF of I/O latency. (b) Average I/O latency.

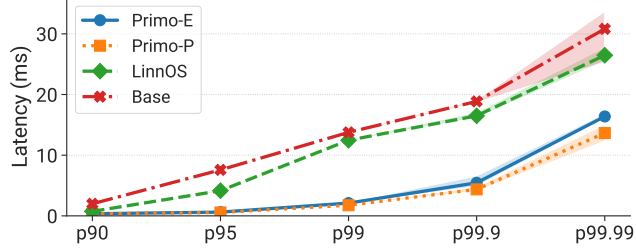


Figure 5: Tail percentiles of I/O latency.

(*Local interpretation*). We can also obtain intuitive cognition of the overall model behavior (*Global interpretation*) through observing the thickness of each decision path (arrow lines).

Specifically, the top-2 layers of the DT show PRIMO first classifies I/O requests from the *current queue length* (L_c), indicating this feature can significantly affect the prediction results. Developers can perform adjustments to L_c thresholds to optimize system behavior. Because the 4-layer DT only contains 7 leaves (terminal nodes), each prediction needs to take at most 4 condition tests at the branch nodes and the majority of test instances only need to execute 2 condition tests. This inference overhead is much smaller than the original DNN model with 8706 parameters in LinnOS. Moreover, as shown in Figure 3 (right), PRIMO only takes 3 input features without any preprocessing, which further reduces the model complexity and deployment overhead. On the contrary, LinnOS needs to perform input data preprocessing for all 9 metrics to form a 31-dimensional input feature (e.g., $L_c = 15$ needs to be converted into a $\{0, 1, 5\}$ vector). This operation is necessary for *every* I/O read operation, remarkably exacerbating the inference overhead.

4.2 Performance Analysis

We evaluate the performance of PRIMO in the LinnOS flash storage I/O scenario from the following two perspectives:

Overall performance. Figure 4 shows the Cumulative Distribution Function (CDF) and average I/O latency (with the standard deviation) of each method over three independent experiments. It is obvious that both PRIMO-E and PRIMO-P significantly outperform LinnOS. Compared with the base I/O mechanism, LinnOS reduces 26.2% I/O latency on average, while PRIMO decreases the I/O latency by 70.0~73.6%. It indicates PRIMO can achieve an additional 2.5~2.8× (PRIMO-E

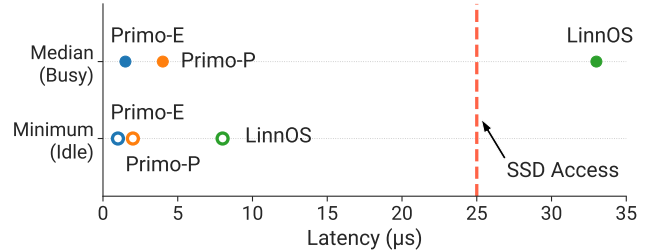


Figure 6: Model inference latency. Empty circles represent the minimum inference latency when the system is idle. Solid circles represent the inference latency of the median I/O operation when the system is busy. The vertical line indicates the basic SSD access latency (reading 4KB data in the idle state).

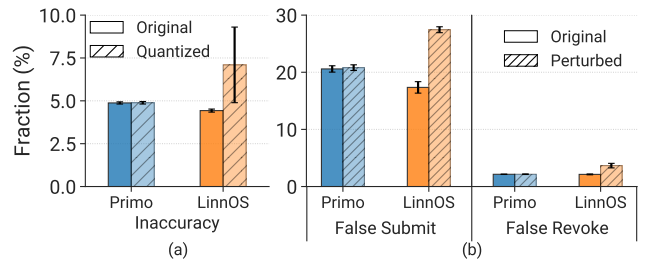


Figure 7: (a) Quantization impact. (b) Robustness test.

/ PRIMO-P) improvement over LinnOS.

Tail performance. The tail behavior is critical to system performance. Figure 5 presents the average I/O latency and the range at tail percentiles (from p90 to p99.99). We find LinnOS fails to reduce tail latency on the tail, and the curve almost overlaps with the Base case. On the contrary, PRIMO-P achieves 7.9×, 4.3× and 2.3× performance improvement over the vanilla I/O mechanism at p99, p99.9 and p99.99 respectively. Additionally, PRIMO-P also performs much better at p90 (2.2×) and p95 (7.5×) compared to LinnOS.

4.3 Effectiveness Analysis

We perform the effectiveness analysis from the following perspectives to investigate why PRIMO can outperform LinnOS.

Inference overhead. In Figure 6, we measure the extra inference latency of PRIMO and LinnOS. (1) When the system is idle, we measure the minimum inference latency. We observe that LinnOS takes 8μs, while the overhead of PRIMO-E is almost negligible ($\leq 1\mu s$), making the deployment more lightweight. (2) When the system is busy with heavy I/O operations, LinnOS requires a median inference latency of 33μs due to the high frequency of preprocessing and inference. This is even higher than the basic SSD access latency (25μs). In contrast, PRIMO remains relatively lower inference latency with smaller overhead.

Quantization. Since floating points are not well supported in the Linux kernel, the model weights of LinnOS and the thresholds of PRIMO are converted to integers by quantization. This can achieve smaller inference latency at the cost of accuracy degradation. Figure 7 (a) shows the quantization

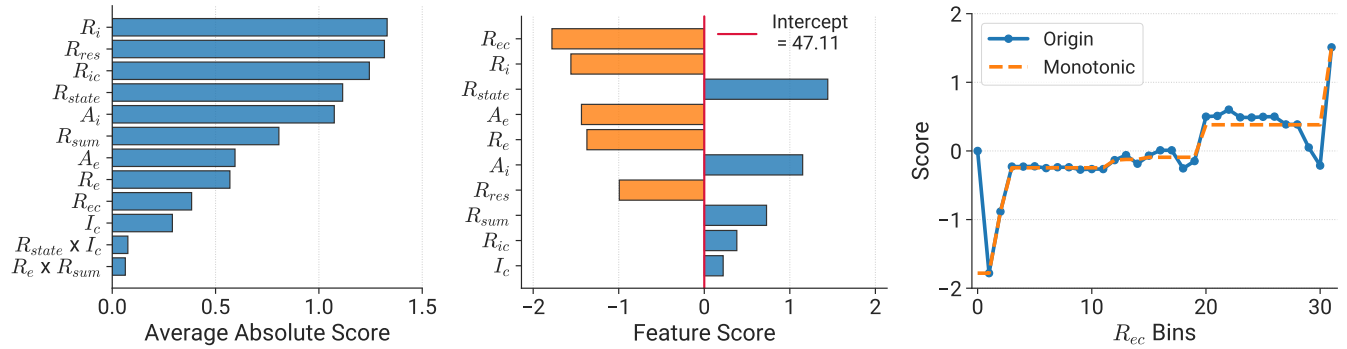


Figure 8: Interpretation and visualization of the PrAM model in Clara-MS. **(Left)**: Global interpretation of overall feature importance. **(Middle)**: Local interpretation of each feature’s contribution to individual predictions. **(Right)**: Visualization of the learned shape function of R_{ec} (blue line), and with the monotonic constraint post-processing optimization (orange line).

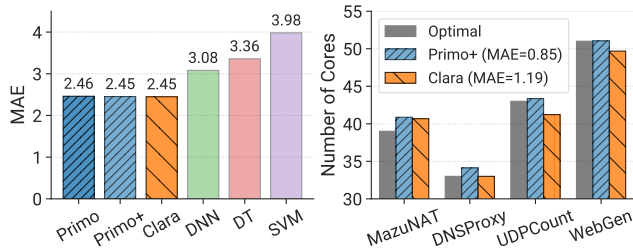


Figure 9: Evaluation on Clara-MS. **(Left)**: Mean Absolute Error (MAE) of testset. **(Right)**: Prediction of 4 real NFs.

impact on the prediction accuracy. It is evident that the accuracy drop of LinnOS is over 2% and varies significantly among different SSD models. In comparison, PRIMO-E has negligible accuracy degradation, as the node threshold values are naturally integers or the decimal part is 0.5.

Robustness. A good model should exhibit high robustness against system state drifting. To measure the robustness of those methods, we synthesize some perturbed samples by adding Gaussian noise to the test dataset. The noise is added to all 4 recent I/O queue lengths ($\sigma = 5$) and I/O latency ($\sigma = 100$)². Figure 7 (b) illustrates the false submit and false revoke rates of LinnOS and PRIMO-E under the original and perturbed test datasets. Reducing the false submit rate is far more important since the *failover* overhead of false revoke is negligible. It is obvious that PRIMO keeps stable accuracy under perturbed input while LinnOS presents severe performance degradation. The robustness of the interpretable model in PRIMO derives from fewer input features and the inherent stability of the tree structure compared with the DNN model.

5 Case Study 2: Clara

Clara [66] is an offline tool that generates offloading insights for network functions (NFs) on SmartNICs. It can analyze a legacy NF in its unported form and suggest the optimal offloading strategies. The main challenge of adopting those ML techniques in Clara is that *insufficient* SmartNIC pro-

²Since the current queue length L_c is the most significant feature, we do not modify its value to avoid changing the real label.

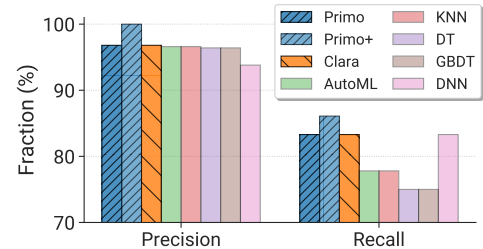


Figure 10: Model precision and recall rates in Clara-AI.

grams can be served to produce training data. Clara has to utilize YarpGen [45] to generate abundant synthesized programs. Clara contains several components for the generation of different offloading insights. Each component adopts a ML algorithm as described below:

- **Multicore Scale-out analysis (Clara-MS).** SmartNICs use multicore parallelism to improve packet processing performance. Clara adopts *GBDT* [17] to predict the optimal number of cores for each NF.
- **Algorithm Identification (Clara-AI).** Certain packet processing algorithms in the host NF can benefit from ASIC accelerators in the SmartNIC. Clara adopts *SVM* [73] to identify such code blocks.
- **Cross-platform Prediction (Clara-CP).** Clara trains an *LSTM* network [9] to predict the number of compute and memory instructions that a NF can be compiled to.

We employ the PrAM model to replace all the three ML models in Clara, as it has better accuracy than PrDT. To analyze the effectiveness of transparent model adjustment in PRIMO, we also evaluate the model performance with post adjustment (denote as PRIMO+). We compare PRIMO with the original models in Clara (LSTM, GBDT and SVM), as well as some alternative baseline algorithms (CNN, DT, TPOT [64] (namely AutoML), K-Nearest Neighbor (kNN)). More details about Clara and our implementation can be found in Appendix D.2.

5.1 System Interpretation

As shown in Figure 8, PRIMO provides comprehensive interpretation for the Clara-MS task, including global and local

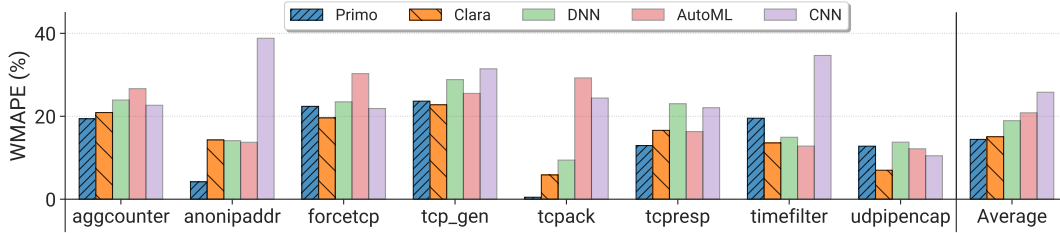


Figure 11: Weighted mean-absolute percentage error (WMAPE) over 8 types of NFs in Clara-CP.

interpretation, as well as transparent shape functions. We list the notation descriptions in Appendix D.2. From the left figure, we find R_i , R_{res} and R_{ic} are the most important features that contribute most to model prediction. Developers should pay more attention to shape function optimizations for these features. We also notice the impact of feature interactions is relatively less important, indicating that we can reduce their priority in model optimization. The middle figure presents the interpretation of the individual prediction for *UDPCount* NF. The final prediction equals the sum of every feature score and the intercept constant (Equation 1). Through the local interpretation, developers can clearly check the model behavior for each prediction to make the corresponding adjustment. Moreover, the right figure (blue line) illustrates the learned shape function for R_{ec} , which allows developers to dive deeper into fine-grained model adjustment (such as the orange line).

5.2 Performance Analysis

Since Clara is an offline system, for each task, we mainly evaluate the model accuracy rather than the inference cost.

Clara-MS. As shown in Figure 9 (left), our interpretable model in PRIMO achieves similar accuracy as the GBDT (XGBoost [17]) model in Clara, and outperforms other ML models over the synthesized test dataset. Figure 9 (right) further presents the accuracy of PRIMO for 4 real NFs. Compared to Clara, PRIMO achieves $1.4\times$ less prediction errors and at most 5% error to the optimal configurations.

Clara-AI. In Figure 10, PRIMO achieves the equivalent precision and recall rates as the SVM model in Clara, and beats other ML algorithms. Through successfully identifying CRC-based NFs, PRIMO could improve peak throughput by $1.6\times$ and decrease latency by 25% [66].

Clara-CP. Clara uses the LSTM model to predict the number of instructions for unported codes. Figure 11 shows the accuracy of the Clara-CP task over 8 representative real NFs and the Average column represents the WMAPE results across all the NFs. We observe PRIMO (14.4%) delivers better performance than Clara (15.1%). This demonstrates the capability of PRIMO to cope with complex program embeddings.

5.3 Model Adjustment

To overcome the training data insufficiency issue, Clara uses YarpGen [45] to generate synthesized programs. This inevitably introduces certain data distribution drifts from the

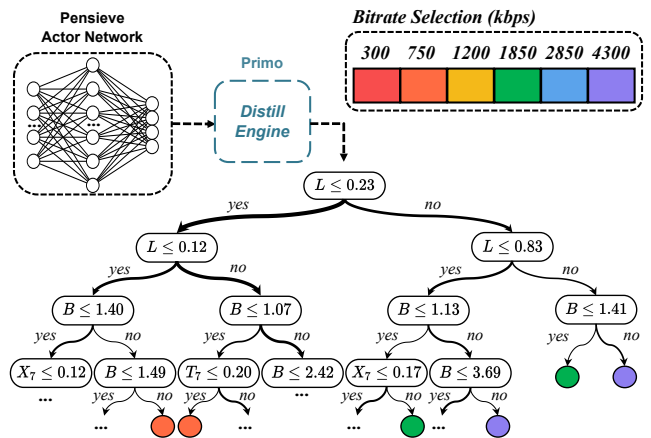


Figure 12: Visualization of the interpretable model distilled from the Pensieve policy. For simplicity, we only present the top 5 layers, and the ellipsis indicates subsequent nodes.

actual scenario. Specifically, there exist instruction distribution differences (0.0303 of Jensen-Shannon divergence and 0.0354 of Bhattacharyya distance) between real-world and synthesized click programs [66]. Such drifts could compromise the model performance. The transparency of the PRIMO model allows developers to discover and fix undesirable behaviors caused by the synthesized data. In addition, PRIMO designs two post-processing tools to help developers adjust the models based on their domain knowledge:

Monotonic Constraint. As introduced in §3.4.1, developers can leverage PRIMO to generate a new shape function with monotonic constraint and rectify the incorrect behaviors of the models automatically. For instance, in Figure 8 (right), the developers know the desired number of cores should be proportional to the memory/compute intensity, i.e., R_{ec} (EMEM/Compute Ratio). Then they can replace the original shape function (blue line) with the monotonic shape function (orange line). They can check each shape function and decide whether it is necessary to apply such adjustment based on their prior knowledge. To evaluate the effectiveness of this strategy, we apply the *Monotonic Constraint* tool to two shape functions (A_i & R_{ec}) and yield the adjusted model PRIMO+. As shown in Figures 9, PRIMO+ achieves better prediction accuracy. This shows the monotonicity of the PRIMO model can be achieved via simple post-processing and appropriate adjustment can bring better performance.

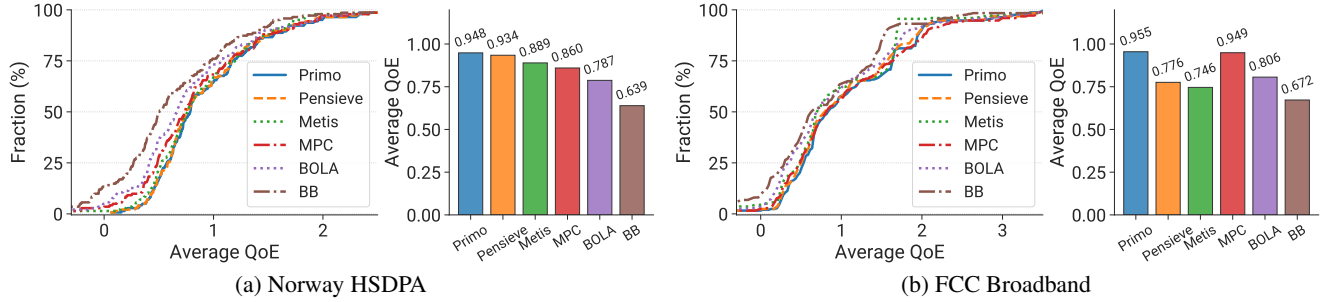


Figure 13: Overall performance of PRIMO compared with other methods on the Norway HSDPA and FCC Broadband traces.

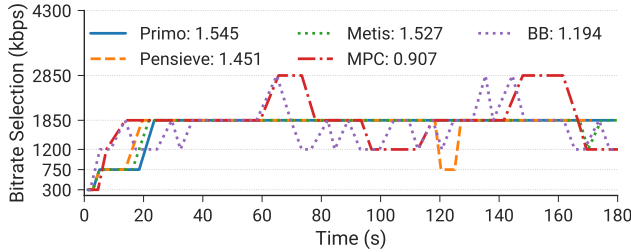


Figure 14: Profiling the bitrate selections of ABR algorithms over one typical Norway HSDPA trace. Legend presents the average QoE of each algorithm.

Counterfactual Explanation. This tool aims to provide simple and intuitive explanations for model troubleshooting. More concretely, it helps developers to understand why this prediction is wrong and how to adjust the model to fix it. We use Clara-AI as an example to describe its usage and evaluation. Clara employs Sequential Pattern Extraction (SPE) [21] to extract code features as *boolean* sequences (each containing 102 features) to indicate whether the NF program contains code blocks for acceleration. Our PRIMO model allocates a contribution score for each feature. To fix False Negative (FN) predictions, we utilize this tool to find the closest k instances from the data set with the opposite label. Through the comparison of these instances, we can easily discover the feature with inadvisable learned scores and adjust the score. In this case, we increase the contribution weight of the 84th feature appropriately. We can also perform transparent debugging for False Positive (FP) predictions similarly. In Figure 10, PRIMO+ further enhances the F1 score from 89.6% to 92.5%.

6 Case Study 3: Pensieve

Our third case study is Pensieve [51], a system that uses RL for online video streaming. It learns the adaptive bitrate (ABR) algorithms automatically to optimize the user quality of experience (*QoE*) defined in Equation 4 in Appendix D.3.

We obtain an interpretable PrDT model through distilling from the original RL actor model. Then we implement the PrDT model into the ABRController of dash.js [2]. More details can be found in Appendix D.3. For baselines, we compare PRIMO with the following algorithms: (1) The RL model in Pensieve. (2) Buffer-Based (BB) [33]: selecting bitrates with the goal of keeping the buffer occupancy above 5 seconds.

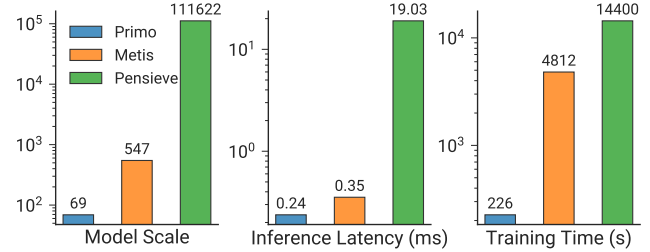


Figure 15: Comparing three learning-based ABR methods.

(3) BOLA [78]: selecting bitrates with Lyapunov optimization on buffer occupancy observations. (4) MPC [87]: selecting bitrates with a control-theoretic model. We evaluate robustMPC variant which can better handle errors in throughput prediction. (5) Metis [55]: using a decision tree to explain the Pensieve RL model, which represents the handcrafted DT approach. Evaluations are performed on the simulator provided by Pensieve, except the deployment experiment (latency).

6.1 System Interpretation

Figure 12 illustrates the learning process with the Distill Engine, as well as the decision making process of the interpretable policy. Related notations are described in Appendix D.3. This DT contains 8 layers and 35 leaves in total, which is compact and simple enough for developers to understand its complete operation logic.

Similar to the PRIMO model in LinnOS (Figure 3), the first 2 layers divide decision flows based on the feature L (Last chunk bitrate) which is in line with our perception. In the third layer, PRIMO proceeds to classify environment states (inputs) according to the feature B (Current buffer size). These observations indicate both L and B are the key features that affect the final bitrate decision, inspiring developers to pay more attention to them when designing ABR algorithms.

6.2 Performance Analysis

Overall performance. Since the ABR algorithm could encounter unprecedented network conditions by different clients, it is important to evaluate its generalization ability. So in addition to the *Norway HSDPA* trace used for model training, we also evaluate another trace *FCC Broadband* that is never applied for training. Figure 13 presents the QoE distribution and average QoE of each method on the two traces. For Nor-

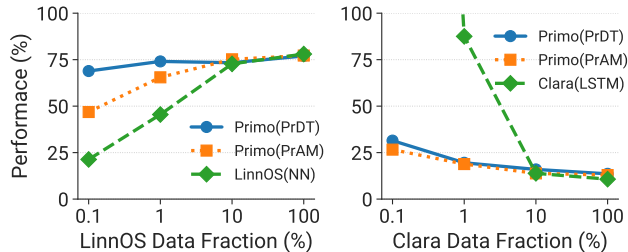


Figure 16: Model performance with less training data. **(Left)**: Recall rate in LinnOS (the higher the better). **(Right)**: WMAPE in Clara-CP (the lower the better).

way HSDPA, the CDF curve of PRIMO almost overlaps with Pensieve’s curve, and the average QoE is even 1.5% higher than Pensieve. This demonstrates PRIMO has successfully learned the Pensieve policy with a simple decision tree and outperforms other ABR algorithms. Furthermore, for FCC Broadband, PRIMO presents better generalization than other two learning-based algorithms. Such advantages are attributed to the adaptive pruning strategy in Bayes Optimization, and the imitation process in the Distill Engine. In contrast, although Metis also uses a decision tree to get a surrogate model from Pensieve, it has some performance degradation as its inflexible pruning strategy.

Example analysis. Figure 14 profiles the bitrate selection actions of different ABR algorithms over a single network trace. We find two heuristic algorithms (BB and MPC) keep fluctuating during the video streaming, which could cause a terrible user experience. The other three learning-based algorithms have more stable decisions. Pensieve decides to decrease the bitrate at 120s and Metis chooses to reduce the video resolution at 170s. This can cause an unsmooth experience (as the penalty term in Equation 4) even though they adjust back the bitrate quickly. In contrast, PRIMO gives a much more smooth experience.

Training and inference overhead. The PRIMO model is more compact and simpler. Figure 15 (left) compares the model complexity³ of different methods. We observe that PRIMO reduces the model scale by 1617× compared to the original Pensieve actor model. Even for Metis which also uses a decision tree, PRIMO can reduce the tree complexity by 7.9×. For inference, PRIMO only needs to perform 3~7 condition tests to make a bitrate decision. It can reduce 70× and 1.5× inference latency compared with Pensieve and Metis respectively, as shown in Figure 15 (middle). To generate a model, in Figure 15 (right), PRIMO only needs less than 4 minutes for model distillation, which is 21.3× faster than Metis (under the same setting). Compare with Pensieve 4 hours training time, less than 4 minutes distill time is ignorable. In summary, PRIMO can greatly reduce overall operating costs in the video streaming scenario.

³Model complexity refers to the number of parameters for Pensieve model, or number of nodes for PRIMO and Metis.

Task	DL Model	Origin	PRIMO	Improvement
LinnOS	3 × DNN (50 epoch)	564s	5s	112.8×
Clara-CP	LSTM (30 epoch)	1,081s	79s	13.7×

Table 3: Training time comparison with original DL models.

Task	Metric	PRIMO w/o BO	PRIMO w/ BO	Improvement
LinnOS	F1 Score	0.8089	0.8518	5.3%
Clara-CP	WMAPE	0.1728	0.1442	16.6%
Clara-MS	MAE	1.0155	0.8660	14.7%

Table 4: Ablation study for Bayes Optimization.

7 More Evaluation

We run some experiments to further evaluate the benefits of PRIMO more comprehensively.

Requiring less training data. Due to the simpler model structure and fewer parameters, PRIMO can have better performance in some scenarios without enough training data like Clara. In Figure 16, we compare the performance of two PRIMO models with the original DL models in LinnOS and Clara-CP using less training data. We use a smaller dataset (10%) in LinnOS as the baseline. Because LinnOS provides abundant data for DNN model training, 10% of original data can provide equivalent performance. It is clear that PRIMO models maintain better performance with limited training data, especially for the PrDT model. Conversely, the original DL models only work with abundant data. This shows PRIMO has broader applicability for various scenarios.

Short training time. Table 3 presents the training time of PRIMO and original DL models in LinnOS and Clara-CP, which adopt the default numbers of training epochs in their papers. PRIMO is able to reduce 2-3 orders of magnitude of training time. Even considering the hyperparameter search process, the significant time conservation could maintain since multiple trails can be executed concurrently. Note that GPUs can only provide very limited acceleration (<1.2×) for these two DL models. Additionally, LinnOS requires training a DNN model for *each* SSD and the prototype only considers three SSDs. In a production-level distributed storage system with thousands of SSDs, LinnOS could have a severe scalability issue. In contrast, PRIMO remarkably saves the training cost, making the deployment more feasible in practice.

Impact of BO. We further perform an ablation study on Bayes Optimization in PRIMO. Table 4 summaries the performance of the PRIMO model with and without BO in LinnOS and Clara. We observe 5.3%~16.6% accuracy improvement brought by BO. Besides, BO typically simplifies the model scale while obtaining better performance. For LinnOS, BO reduces over 15× tree nodes compared with PRIMO without BO. This verifies the importance of the BO component in making interpretable models more practical. For search time aspect, BO obtains over 1.2× acceleration compared with naive random search algorithm in Clara-MS task by reducing search trails to reach equivalent performance.

8 Discussion

Is the interpretation always correct? Yes. PRIMO provides the interpretation correctness guarantee for each generated model and each prediction. Existing interpretation tools [39, 48, 67] aim to offer explanations for understanding black-box models, whereas the generated interpretations are sometimes contradictory or even mislead users. In contrast, PRIMO models are inherently interpretable and developers can totally trust the interpretation.

Can PRIMO be applied to all systems? PRIMO has its limitations in some system scenarios. For instance, it does not yet support unsupervised learning scenarios (e.g., anomaly detection in security applications [16]). It cannot outperform black-box models in systems with extremely complex features, e.g., images, speeches. These will be our future work.

Is the post-processing step necessary? These operations are optional because the trained models without post-processing usually have excellent performance. In order to take full advantage of the interpretable models, the post-processing tools help developers leverage their expertise and domain knowledge to further optimize system performance. In a black-box model, it is hard to perform such optimization.

Can PRIMO work on a larger model? Yes. We have demonstrated PRIMO can outperform DNN models in various scenarios, including LinnOS (MLP with 8×10^3 parameters), Clara-CP (LSTM+FC with 4×10^4 parameters) and Pensieve (CNN+MLP with 1×10^5 parameters). They represent most model scale of learning augmented systems listed in [1]. For larger models, we evaluate Habitat [88] as an example. It leverages 8-layer MLP models, containing over 8×10^6 parameters, to prediction DL operation execution time on heterogeneous GPUs. PRIMO can provide comparable prediction accuracy as Habitat across conv2d, linear, lstm and bmm operations.

How to interpret high-dimensional data? Admittedly, when handling high-dimensional datasets, PRIMO models may become more complicated for users to understand the whole model. However, PRIMO provides ordered feature importance for interpretation. Generally, users can focus on the top several tree layers of PrDT or several significant shape functions according to the global interpretation of PrAM.

Future work. There are four possible directions as our future work. (1) We can extend PRIMO to support learning-augmented systems with unsupervised learning. (2) To obtain a more accurate interpretable model, we can optimize the model training algorithm. Currently, we use CART [14], the most popular and widely applied approach, for decision tree learning in PrDT. This is based on the heuristic greedy algorithm where locally optimal decisions are made in each node. In the future, we plan to employ novel decision tree training algorithms (e.g., GOSDT [44], based on dynamic programming method) to solve the sub-optimal problem. (3) For practical deployment, comprehensive programming language support is needed because different systems have their own coding

requirements. PrDT has a tool for converting Python-based models to other formats. But PrAM only supports the conversion to the ONNX [4] format currently. We aim to provide more model format conversion in the future. (4) Integration with existing RL-based system development frameworks (e.g., Park [50]) to facilitate more practical system deployment.

9 Related Work

Interpretability of learning-augmented systems. To our best knowledge, there is no prior work that develops a unified framework for providing inherent interpretability for systems like PRIMO. Besides, interpretability is often overlooked during the development of learning-augmented systems, and only a few works consider it. Bao [53] is a learning-based system that adopts TreeCNN [60] for query optimization and the decision process can be inspected by developers. Tang et al. [79] proposed an interpretable method that extracts a Finite State Machine from a RL policy for storage resource allocation in Huawei. Grüner et al. [25] generated concise and interpretable rule-sets for unknown proprietary streaming algorithms (e.g. ABR approaches in Youtube, Twitch), which is similar to the Pensieve case study with PRIMO (§6).

Some efforts were also made on building tools for interpreting black-box models [26, 27, 55], as discussed in §2.3.1. Different from these methods, PRIMO does not seek for interpreting black-box models but directly building transparent models, with higher fidelity and efficiency.

Machine learning and system co-design. It is non-trivial to apply ML techniques for system design and deployment in practice. Autosys [42] introduces a framework to address common design considerations (e.g., learning-induced system failures, extensibility), and reported years of experiences in designing and operating learning-augmented systems at Microsoft. WhiRL [19] facilitates the safe deployment of RL-based systems through verifying whether the learned policy meets the designer's requirements. Some components in PRIMO are inspired from these works.

10 Conclusion

This work introduces PRIMO, a unified framework that assists developers to design practical learning-augmented systems with interpretable models. For different scenarios, PRIMO provides respective models and optimization solutions to meet the system requirements. Based on our case studies, we demonstrate that PRIMO can achieve *transparent, accurate and lightweight* system deployment in practice.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their valuable comments and suggestions. This study is supported under the RIE2020 Industry Alignment Fund–Industry Collaboration Projects (IAF-ICP) Funding Initiative, as well as cash and in-kind contributions from the industry partner(s).

References

- [1] Awesome-ml-for-system. <https://github.com/S-Lab-System-Group/Awesome-ML-for-System>, 2022.
- [2] Dash.js: Javascript player. <https://github.com/Dash-Industry-Forum/dash.js>, 2022.
- [3] Federal communications commission. <https://www.fcc.gov/reports-research/reports/measuring-broadband-america>, 2022.
- [4] Onnx: Open neural network exchange. <https://github.com/onnx/onnx>, 2022.
- [5] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing slos for resource-harvesting vms in cloud platforms. OSDI '20.
- [6] Miriam Ayer, H. D. Brunk, G. M. Ewing, W. T. Reid, and Edward Silverman. An empirical distribution function for sampling with incomplete information. *The Annals of Mathematical Statistics*, 1955.
- [7] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? NeurIPS '14.
- [8] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman Amarasinghe. A deep learning based cost model for automatic code optimization. MLSys '21.
- [9] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. ICLR '15.
- [10] Erick Carvajal Barboza, Sara Jacob, Mahesh Ketkar, Michael Kishinevsky, Paul Gratz, and Jiang Hu. Automatic microprocessor performance bug detection. HPCA '21.
- [11] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. NeurIPS '18.
- [12] Shane Bergsma, Timothy Zeyl, Arik Senderovich, and J. Christopher Beck. Generating complex, realistic cloud workloads using recurrent neural networks. SOSP '21.
- [13] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 2012.
- [14] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. Classification and regression trees. 1984.
- [15] Nilotpal Chakravarti. Isotonic median regression: A linear programming approach. *Mathematics of Operations Research*, 1989.
- [16] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 2009.
- [17] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. KDD '16.
- [18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. USENIX ATC '19.
- [19] Tomer Eliyahu, Yafim Kazak, Guy Katz, and Michael Schapira. Verifying learning-augmented systems. SIGCOMM '21.
- [20] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. ICML '18.
- [21] Yujie Fan, Yanfang Ye, and Lifei Chen. Malicious sequential pattern mining for automatic malware detection. *Expert Systems with Applications*, 2016.
- [22] Keinosuke Fukunaga and Patrenahalli M. Narendra. A branch and bound algorithms for computing k-nearest neighbors. *IEEE Transactions on Computers*, 1975.
- [23] Yuanxiang Gao, Li Chen, and Baochun Li. Spotlight: Optimizing device placement for training deep neural networks. ICML '18.
- [24] Liangyi Gong, Zhenhua Li, Feng Qian, Zifan Zhang, Qi Alfred Chen, Zhiyun Qian, Hao Lin, and Yunhao Liu. Experiences of landing machine learning onto market-scale mobile malware detection. EuroSys '20.
- [25] Maximilian Grüner, Melissa Licciardello, and Ankit Singla. Reconstructing proprietary video streaming algorithms. USENIX ATC '20.
- [26] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. Lemna: Explaining deep learning based security applications. CCS '18.
- [27] Dongqi Han, Zhiliang Wang, Wenqi Chen, Ying Zhong, Su Wang, Han Zhang, Jiahai Yang, Xingang Shi, and Xia Yin. Deepaid: Interpreting and improving deep learning-based anomaly detection in security applications. CCS '21.

- [28] Xueyuan Han, Xiao Yu, Thomas Pasquier, Ding Li, Junghwan Rhee, James Mickens, Margo Seltzer, and Haifeng Chen. SIGL: Securing software installations through deep graph learning. *USENIX Security* '21.
- [29] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. Linnos: Predictability on unpredictable flash storage with a light neural network. *OSDI* '20.
- [30] Trevor Hastie and Robert Tibshirani. Generalized additive models: Some applications. *Journal of the American Statistical Association*, 1987.
- [31] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *CoRR*, 2015.
- [32] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. *SC* '21.
- [33] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: evidence from a large video streaming service. *SIGCOMM* '14.
- [34] Sarthak Jain and Byron C. Wallace. Attention is not explanation. *NAACL* '19.
- [35] José Jiménez-Luna, Francesca Grisoni, and Gisbert Schneider. Drug discovery with explainable artificial intelligence. *Nature Machine Intelligence*, 2020.
- [36] Michael I. Jordan and Robert A. Jacobs. Hierarchical mixtures of experts and the em algorithm. *IJCNN* '93.
- [37] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *NeurIPS* '17.
- [38] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbah, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. *USENIX ATC* '20.
- [39] Narine Kokhlikyan, Vivek Miglani, Miguel Martin, Edward Wang, Bilal Alsallakh, Jonathan Reynolds, Alexander Melnikov, Natalia Kliushkina, Carlos Araya, Siqi Yan, and Orion Reblitz-Richardson. Captum: A unified and generic model interpretability library for pytorch. *CoRR*, 2020.
- [40] Mikel Landajuela, Brenden K Petersen, Sookyung Kim, Claudio P Santiago, Ruben Glatt, Nathan Mundhenk, Jacob F Pettit, and Daniel Faissol. Discovering symbolic policies with deep reinforcement learning. *ICML* '21.
- [41] Xu Li, Feilong Tang, Jiacheng Liu, Laurence T. Yang, Luoyi Fu, and Long Chen. AUTO: Adaptive congestion control based on multi-objective reinforcement learning for the satellite-ground integrated network. *USENIX ATC* '21.
- [42] Chieh-Jan Mike Liang, Hui Xue, Mao Yang, Lidong Zhou, Lifei Zhu, Zhao Lucis Li, Zibo Wang, Qi Chen, Quanlu Zhang, Chuanjie Liu, and Wenjun Dai. Autosys: The design and operation of learning-augmented systems. *USENIX ATC* '20.
- [43] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for in-storage data retrieval. *USENIX ATC* '19.
- [44] Jimmy Lin, Chudi Zhong, Diane Hu, Cynthia Rudin, and Margo Seltzer. Generalized and scalable optimal sparse decision trees. *ICML* '20.
- [45] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpgen. *Proceedings of the ACM on Programming Languages*, 2020.
- [46] Yin Lou, Rich Caruana, and Johannes Gehrke. Intelligent models for classification and regression. *KDD* '12.
- [47] Yin Lou, Rich Caruana, Johannes Gehrke, and Giles Hooker. Accurate intelligible models with pairwise interactions. *KDD* '13.
- [48] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *NeurIPS* '17.
- [49] Nguyen Cong Luong, Dinh Thai Hoang, Shimin Gong, Dusit Niyato, Ping Wang, Ying-Chang Liang, and Dong In Kim. Applications of deep reinforcement learning in communications and networking: A survey. *IEEE Communications Surveys Tutorials*, 2019.
- [50] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Ravichandra Addanki, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, Frank Cangialosi, Shaileshh Bokka Venkatakrishnan, Wei-Hung Weng, Song Han, Tim Kraska, and Mohammad Alizadeh. Park: An open platform for learning-augmented computer systems. *NeurIPS* '19.
- [51] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. *SIGCOMM* '17.

- [52] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. *SIGCOMM '19*.
- [53] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. *SIGMOD '21*.
- [54] Zili Meng, Yaning Guo, Yixin Shen, Jing Chen, Chao Zhou, Minhu Wang, Jia Zhang, Mingwei Xu, Chen Sun, and Hongxin Hu. Practically deploying heavyweight adaptive bitrate algorithms with teacher-student learning. *IEEE/ACM Transactions on Networking*, 2021.
- [55] Zili Meng, Minhu Wang, Jiasong Bai, Mingwei Xu, Hongzi Mao, and Hongxin Hu. Interpreting deep learning-based networking systems. *SIGCOMM '20*.
- [56] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. *ICML'17*.
- [57] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *ICML '16*.
- [58] Christoph Molnar. *Interpretable Machine Learning*. 2019.
- [59] Shanka Subhra Mondal, Nikhil Sheoran, and Subrata Mitra. Scheduling of time-varying workloads using reinforcement learning. *AAAI '21*.
- [60] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. *AAAI '16*.
- [61] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. Ofc: An opportunistic caching system for faas platforms. *EuroSys '21*.
- [62] Harsha Nori, Rich Caruana, Zhiqi Bu, Judy Hanwen Shen, and Janardhan Kulkarni. Accuracy, interpretability, and differential privacy via explainable boosting. *ICML '21*.
- [63] Harsha Nori, Samuel Jenkins, Paul Koch, and Rich Caruana. Interpretml: A unified framework for machine learning interpretability. *CoRR*, 2019.
- [64] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. *GECCO '16*.
- [65] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices. *OSDI '20*.
- [66] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated smartnic offloading insights for network functions. *SOSP '21*.
- [67] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier. *KDD '16*.
- [68] Haakon Riiser, Paul Vigmostad, Carsten Griwodz, and Pål Halvorsen. Commute path bandwidth traces from 3g networks: analysis and applications. *MMSys '13*.
- [69] Aaron M. Roth, Nicholay Topin, Pooyan Jamshidi, and Manuela Veloso. Conservative q-improvement: Reinforcement learning for an interpretable decision-tree policy. *CoRR*, 2019.
- [70] Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 2019.
- [71] Cynthia Rudin, Chaofan Chen, Zhi Chen, Haiyang Huang, Lesia Semenova, and Chudi Zhong. Interpretable machine learning: Fundamental principles and 10 grand challenges. *CoRR*, 2021.
- [72] Cynthia Rudin and Berk Ustun. Optimized scoring systems: Toward trust in machine learning for healthcare and criminal justice. *INFORMS Journal on Applied Analytics*, 2018.
- [73] Bernhard Schölkopf, Alex J. Smola, Robert C. Williamson, and Peter L. Bartlett. New Support Vector Algorithms. *Neural Computation*, 2000.
- [74] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. *USENIX Security '15*.
- [75] Andrew Silva, Matthew Gombolay, Taylor Killian, Ivan Jimenez, and Sung-Hyun Son. Optimization methods for interpretable differentiable decision trees applied to reinforcement learning. *AISTATS '20*.
- [76] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. *NeurIPS '12*.

- [77] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. NSDI '20.
- [78] Kevin Spiteri, Rahul Uргаonkar, and Ramesh K. Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. INFOCOM '16.
- [79] Yingtian Tang, Han Lu, Xijun Li, Lei Chen, Mingxuan Yuan, and Jia Zeng. Learning-aided heuristics design for storage system. SIGMOD '21.
- [80] Arnaud Van Looveren and Janis Klaise. Interpretable counterfactual explanations guided by prototypes. ECML-PKDD '21.
- [81] Jiachen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. Polyjuice: High-performance transactions via learned concurrency control. OSDI '21.
- [82] Xingda Wei, Rong Chen, and Haibo Chen. Fast rdma-based ordered key-value store using remote learned cache. OSDI '20.
- [83] Sarah Wiegrefe and Yuval Pinter. Attention is not not explanation. EMNLP '19.
- [84] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. ICML '15.
- [85] Yong Xu, Kaixin Sui, Randolph Yao, Hongyu Zhang, Qingwei Lin, Yingnong Dang, Peng Li, Keceng Jiang, Wenchi Zhang, Jian-Guang Lou, Murali Chintalapati, and Dongmei Zhang. Improving service availability of cloud systems by predicting disk error. USENIX ATC '18.
- [86] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H. Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. SoCC '17.
- [87] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. SIGCOMM '15.
- [88] Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. Habitat: A runtime-based computational performance predictor for deep neural network training. USENIX ATC '21.
- [89] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. Rlscheduler: An automated hpc batch job scheduler using reinforcement learning. SC '20.
- [90] Ji Zhang, Ping Huang, Ke Zhou, Ming Xie, and Sebastian Schelter. Hddse: Enabling high-dimensional disk state embedding for generic failure detection system of heterogeneous disks in large data centers. USENIX ATC '20.
- [91] Xinyang Zhang, Ningfei Wang, Hua Shen, Shouling Ji, Xiapu Luo, and Ting Wang. Interpretable deep learning under fire. USENIX Security '20.
- [92] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. Sinan: MI-based and qos-aware resource management for cloud microservices. ASPLOS '21.
- [93] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Anso: Generating high-performance tensor programs for deep learning. OSDI '20.
- [94] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. ASPLOS '20.
- [95] Hang Zhu, Varun Gupta, Satyajeet Singh Ahuja, Yuan-dong Tian, Ying Zhang, and Xin Jin. Network planning with deep reinforcement learning. SIGCOMM '21.
- [96] Kostas Zoumpatianos, Yin Lou, Ioana Ileana, Themis Palpanas, and Johannes Gehrke. Generating data series query workloads. *The VLDB Journal*, 2018.

A Supplementary Elaborations

In this section, we provide some supplementary elaborations about PRIMO algorithms, including algorithm detailed illustrations of PrAM and PrDT.

A.1 PrAM: Explainable Boosting Machine

Explainable Boosting Machine (EBM) [63] is a open-source implementation of Generalized Additive Models plus Interactions (GA²M) [47] written in C++ and Python. Similar to popular GBDT algorithms (e.g. LightGBM [37]), the EBM training procedure begins by bucketing data from continuous features into discrete bins of histogram [62], which can significantly accelerate model training. Then EBM starts to learn shape function $f_i(\cdot)$ for each feature. Common choices for shape functions are regression splines, step functions and binary trees. For better prediction accuracy, it chooses the boosted trees, where each successive tree tries to predict the overall residual from all the preceding trees [46]. Furthermore, EBM optimizes the traditional boosting (greedy search) approach by leveraging *cyclic gradient boosting*, which learns a shallow tree for each feature in a round-robin fashion [62]. PrAM is heavily based on the implementation of EBM.

For simplicity and accuracy, PrAM leverages BO to automatically adjust model configurations, including the number of histogram bins, the number of considered interactions, learning rate, etc. It helps developers easily obtain the optimal model which is compact and accurate.

A.2 PrDT: Minimal Cost-Complexity Pruning

We adopt Minimal Cost-Complexity Pruning (CCP) [14] to prune the full tree in the *post-processing* stage. This algorithm aims to minimize a cost-complexity metric $R_\alpha(T)$ which is defined as

$$R_\alpha(T) = R(T) + \alpha \cdot N(T) \quad (3)$$

where $R(T)$ and $N(T)$ denote the misclassification cost (error rate) and complexity penalty (the number of leaves) of the decision tree T respectively. The trade-off between accuracy and sparsity of the tree is controlled by the *complexity parameter* α : as α increases, more leaves are pruned and the total impurity increases.

Figure 17 presents the impact of the *complexity parameter* α on the model performance and complexity for the LinnOS system (§4). When α is too small (before the red dashed line), PrDT has poor performance because of the severe overfitting. With a bigger α , developers could trade-off the model accuracy and complexity based on their system requirements. The optimal pruning factor can differ by many orders of magnitude for different systems according to our experiments. It is hard for system developers to identify an appropriate value of α intuitively. To address this, PRIMO adopts bayes optimization to perform post pruning automatically (§3.3.1).

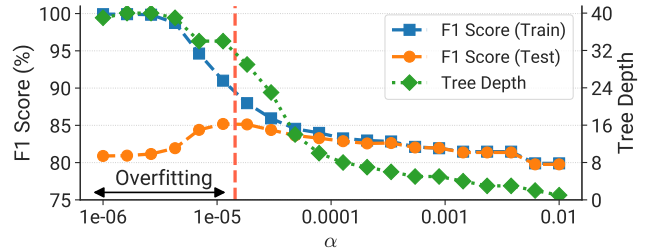


Figure 17: Performance (F1 Score) and complexity (Tree Depth) of the PrDT model under different α in LinnOS.

B Insufficiency of Existing Methods

To demonstrate the argument in §2.3, we provide some examples and perform analysis. These experiments reveal the insufficiency of both existing machine learning frameworks and interpretation tools.

B.1 Contradictory Interpretation

Clara adopts XGBoost to predict the optimal core counts for different NFs in Clara-MS. XGBoost contains a built-in api `get_score` to get the feature importance value of each feature and it can be regarded as the model interpretation. However, we argue that the interpretation has low fidelity. As evident from the Figure 18, interpretations based on different metrics present contrasting patterns, where both *gain* and *cover* are important intermediate metrics during model generation. For instance, R_{sum} is the second important feature according to *cover*-based explanation while seems ignorable from *gain*'s perspective. This makes developers feel confused which interpretation should trust. More seriously, this could mislead them to make wrong decisions, such as incorrectly omitting important input features while feature engineering.

B.2 Incapability for Global Surrogate

We have demonstrated the remarkable performance of PRIMO in this work. A common question is that *If using the existing interpretation tools for the model surrogate, can they also deliver equivalent effects?* Our answer is no. We evaluate two popular black-box interpretation methods: Lime [67] and Lemna [26]. Lime performs local interpretation through linear regression of data subset and Lemna adopts a mixture regression model to obtain interpretation in a similar way. Both of them are designed for local interpretation of several instances. However, we further explore whether they have the potential for the global model surrogate. Hence, we train the Lime and Lemna model on the LinnOS dataset, and compare the performance with PRIMO. As shown in Figure 19, Lime and Lemna cause much higher (7.3%~9.0%) false submit rates, which is the most significant metric for the system performance. As stated in §4, the overhead of failover (false revoke) is relatively negligible and all three methods perform well pertaining to this metric. Overall, existing interpretation tools are insufficient for the global model surrogate.

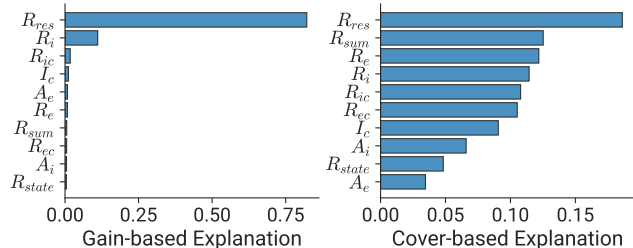


Figure 18: Interpretations of XGBoost model for Clara-MS task based on different metrics. Higher value indicates the feature is more important.

C Lessons Learned From the Case Studies

In order to draw high level conclusions from the three case studies discussed in §4, §5 and §6, we list the key observation of the PRIMO benefits for each system scenario as below. To summarize, in addition to model transparency, PRIMO provides higher prediction accuracy, smaller inference overhead and better model generalization ability. These greatly facilitate model deployment in practice.

1. LinnOS

Key Observation 1

The high inference overhead of the DNN model can hinder its deployment in practice, meanwhile seriously damaging the effect of system performance improvement. PRIMO successfully overcomes this bottleneck.

2. Clara

Key Observation 2

Instead of using various black-box models for different tasks, PRIMO uses a unified interpretable model achieving equivalent even better accuracy and endows capability of model adjustment to remedy the problem caused by drifted synthesis data.

3. Pensive

Key Observation 3

With PRIMO, we obtain an interpretable model that has better performance and generalization ability than the RL policy. Moreover, it achieves much less inference overhead and low distill cost for practical deployment.

D More Details about the Evaluated Systems

D.1 LinnOS

LinnOS infers the SSD speed using a lightweight neural network. The motivation behind LinnOS is that SSD read latency is unstable because some SSD internal operations (e.g., write-triggered garbage collection, buffer flushing) are contending with user read I/Os. In addition, there are the same replicas

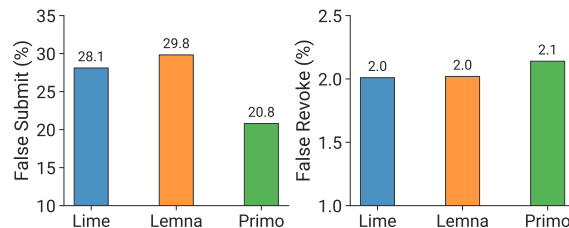


Figure 19: Comparison of Primo global surrogate performance with Lime and Lemna in the LinnOS scenario.

in other SSDs within the storage array (e.g., flash RAID) and we can utilize the built-in *failover* logic to circumvent slow read I/Os. The overhead of switching the *read* operation to a redundant SSD (namely *failover*) is ignorable compared with I/O pending time.

Implementation. We implemented PrDT models into the Linux kernel v5.4.8 (same version with LinnOS) within the *block* layer (written in C). We use the same SSD I/O traces in LinnOS, which were collected from Microsoft Bing Index servers and have been preprocessed by the authors.

Although the LinnOS workflow files are open-sourced on the Chameleon Cloud [38], the experiment results are unreliable due to the unstable and random SSD I/O accesses (also argued by the authors). Consequently, we shift our implementation and evaluation on a bare-metal server from CloudLab [18]. It contains four homogeneous enterprise-level 1.6TB SSDs. One of them serves as the system drive and the rest three SSDs are used for performance evaluation. Additionally, due to the rapid development of the SSD technology in recent years, the Microsoft traces collected in 2016 cannot give sufficient load pressure for evaluation. So we execute a constant writing task in the background for each SSD (LinnOS only records read I/Os). According to our numerous tests, the additional load will not cause fluctuations in the evaluation results.

For the PrDT interpretation results, PRIMO can provide a more precise visualized file of the PrDT which covers the number of samples at each flow and the Gini impurity value of each node rather than Figure 3.

D.2 Clara

Clara is a system that generates the optimal offloading strategies for NFs in SmartNIC. Since the performance characteristics of offloaded programs are opaque prior to porting and offloading strategies are difficult to reason about, developers need to first cross-port NFs to the SmartNIC, perform workload-specific benchmarks, and then iteratively tune the ported programs to achieve higher performance. Clara can analyze a legacy NF in its unported form and suggest porting strategies for the given NF to achieve higher performance.

Implementation. We follow the author-provided data preprocessing pipeline to deal with synthesized and real traces for training and evaluation. We conduct Clara evaluation on a

Notation	Description
A_i	IMEM Access
A_e	EMEM Access
I_c	Compute Intensity
R_i	IMEM / Overall Intensity Ratio
R_e	EMEM / Overall Intensity Ratio
R_{ic}	IMEM / Compute Ratio
R_{ec}	EMEM / Compute Ratio
R_{sum}	(IMEM + EMEM) / Compute Ratio
R_{res}	(IMEM - EMEM) / Compute Ratio
R_{state}	Non-Stateful / Stateful Compute Ratio

Table 5: Notation descriptions of Clara-MS.

Ubuntu 20.04 server with one Intel Core i9-10900 CPU, 64 GB memory and an NVIDIA RTX 2080Ti GPU. Note that because it is an offline system and we focus on model accuracy, Netronome SmartNIC is not required in our experiment.

Notation. We clarify the notations used in system interpretations (Figure 8). Table 5 shows processed features used in Clara-MS, where IMEM indicates SRAM-based internal memory and EMEM indicates DRAM-based external memory on SmartNICs. Instructions can be classified into Stateful (e.g., loads and stores to global variables in memory) and Non-Stateful (e.g., compute instructions, or accesses to function-local variables). Moreover, Overall Intensity represents the sum of IMEM, EMEM and Compute Intensity.

D.3 Pensieve

Pensieve [51] is a system that learns adaptive bitrate (ABR) algorithms automatically using RL technique. The online videos are stored on servers as multiple chunks (a few seconds of the video) and each chunk is encoded at several discrete bitrates (resolutions). Specifically, Pensieve adopts A3C [57] to perform RL training. Both the actor and critic networks use the same NN structure which contains a 1D-CNN layer and a fully connected layer. The actor takes recent network observations as input and suggests the bitrate for the next video chunk as the output. Content providers employ ABR algorithms to optimize user quality of experiment (QoE) which is defined as:

$$QoE = \sum_{n=1}^N q(R_n) - \mu \sum_{n=1}^N T_n - \sum_{n=1}^{N-1} |q(R_{n+1}) - q(R_n)| \quad (4)$$

where R_n represents the bitrate of the n^{th} chunk and $q(R_n)$ maps that bitrate to the quality perceived by a user. T_n represents the rebuffering time and the last term penalizes changes in video quality to favor smoothness.

Implementation. We employ the same server used in Clara for the Pensieve experiment. The video server is based on Apache httpd (Version 2.4.41) and uses Google Chrome (Version 96) as the client video player. We use 142 Norway HSDPA [68] network traces (provided by the authors) for evaluation, in addition, we process another 249 FCC [3, 54] traces

Notation	Description
X_t	Past chunk throughput
T_t	Past chunk download time
N_k	Next chunk sizes
B	Current buffer size
C	Number of chunks left
L	Last chunk bitrate

Table 6: Notation descriptions of Pensieve.

(2018 version, follow the same preprocessing pipeline) for model generalization evaluation. Because the original FCC-18 network speed is much faster than HSDPA and we cannot distinguish the performance difference among algorithms, we scale down the network speed by 4 times to keep consistent with FCC-16 with regards to the median values.

We obtain PrDT model through distilling from the trained RL actor model (pre-trained model that Pensieve authors provided). After that, we implement PrDT (written in JavaScript) into *ABRController* of dash.js [2] (Version 2.4). For the test videos, we use the same video in Pensieve at bitrates in {300, 750, 1200, 1850, 2850, 4300} kbps (which pertain to video modes in {240, 360, 480, 720, 1080, 1440}p). This video is divided into 48 chunks and each chunk represented approximately 4 seconds. Furthermore, we adopt $q(R_n) = R_n$ in Equation 4 to set *linear QoE* as the our evaluation metric.

Notations. We clarify notations used in system interpretations (Figure 12). Table 6 shows environment state variables used in Pensieve to generate bitrate decision, where X_t and T_t denotes past sequences of throughput and download time respectively ($t = 1, \dots, 8$). Moreover, N_k represents sequence of next chunk sizes ($k = 1, \dots, 6$).

E Artifact

PRIMO and case study scripts are available as below. To reproduce the main results of this work, we also provide detailed documentation and instructions in the artifact repository.

Artifact Links

GitHub: <https://github.com/S-Lab-System-Group/Primo>

DOI: <https://doi.org/10.5281/zenodo.6529892>

Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems

Rong Gu Han Yin Weichang Zhong Chunfeng Yuan Yihua Huang
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

Abstract

Stateful distributed stream processing engines (SPEs) usually call for dynamic rescaling due to varying workloads. However, existing state migration approaches suffer from latency spikes, or high resource usage, or major disruptions as they ignore the order of state migration during rescaling. This paper reveals the importance of state migration order to the latency performance in SPEs. Based on that, we propose Mecas, an on-the-fly state migration mechanism which prioritizes the state migration of hot keys (those being processed or about to be processed by downstream operator tasks) to achieve smooth rescaling. Mecas leverages a fetch-on-demand design which migrates operator states at record-granularity for state consistency. We further devise a hierarchical state data structure and gradual strategy for migration efficiency. Mecas is implemented on Apache Flink and evaluated with diversified benchmarks and scenarios. Compared to state-of-the-art approaches, Mecas improves stream processing performance in terms of latency and throughput during rescaling by orders of magnitude, with negligible overhead and no disruption to non-rescaling periods.

1 Introduction

In recent years, stateful Stream Processing Engines (SPEs) [16, 19, 33, 44, 53] have been widely adopted because of the increasing demands for complicated analytics over real-time data, e.g. fraud detection, log monitoring, sentiment analysis, and IoT applications [11, 13, 51, 60].

SPEs are expected to be long-running and always have low latency performance [21, 22]. It commonly requires SPEs to perform dynamic rescaling in the face of unpredictable circumstances (e.g. data rate fluctuations, machine failures, processing stragglers). However, as the processing operators in SPE are usually stateful and partitioned across workers, rescaling them calls for state migration, which means moving state data between workers, even across networks [26].

The problem of state migration in SPEs is fundamental and challenging. Prior major advances made in the last decades

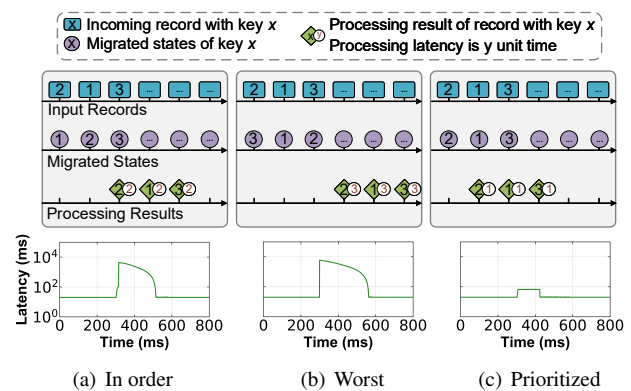


Figure 1: Impact of migration order on processing latency. The Y-axis indicates event-time latency, i.e., interval between a record’s event-time and its emission-time from the output operator [30]. Point (X, Y) means the average processing latency of the records generated at X ms is Y ms.

can be classified into four categories: (1) Full-Restart approach [16, 33, 53] pauses and resumes the whole task when redistributing states. (2) Partial-Pause approach [14, 52] restarts a subgraph instead of the whole job topology to reduce execution blocking. (3) Replicated-Dataflow approach [47, 61] executes a new dataflow in parallel with the old one until finishing the state migration. (4) Proactive approach [26, 43] adds extra behaviour to non-rescaling stream processing periods to relieve the pressure during state migration.

Unfortunately, prior approaches suffer from processing latency spikes, or high resource usage [26], or major disruption (see Section 5.6). Their common limitation lies in **order-unaware** state migration, i.e., not taking into account the order in which operator state migrates. Figure 1 illustrates how the migration order affects SPE latency performance during rescaling. We take a representative stateful stream processing job, key-count, as an example, where records with random keys come from upstream and are processed by the SPE in a FIFO manner. In this case, streaming operators store the

count value of each key as their corresponding states. During rescaling, an affected operator needs to receive the migrated state (the global count values of keys) before it can process the corresponding incoming records.

If the states are migrated in an order-unaware manner as in Figure 1(a), the first coming record may not be processed in time, because it needs to wait for the arrival of its corresponding state. This also blocks subsequent records in the queue due to FIFO processing, eventually accumulating the processing latency for all records over a period of time. In the worst case as shown in Figure 1(b), the state of the first record is the last one to be migrated, making all record processing blocked before the state migration is finished. Ideally, as shown in Figure 1(c), states are migrated in exactly the same order as the records arrive, therefore minimal time is spent in the waiting queue and the latency can be greatly reduced.

Based on this observation, we find that the state of hot keys (those being processed or about to be processed by downstream operator tasks) needs to be prioritized so that the stream processing proceeds without blocking. In this paper, we propose Mecas, an on-the-fly rescaling mechanism which enables **prioritized state migration** for SPEs. In fact, it is challenging to dynamically adjust the state migration order according to the incoming records during SPE rescaling. To achieve this, Mecas leverages a fetch-on-demand state accessing model, based on the fact that modern SPEs [16, 53] co-partition data records with stream operators to the same key space. During rescaling, the states can be actively *fetch*ed by the SPE operator instances when receiving a data record whose state is not local. In this way, the operators prioritize the transmission of those currently needed states to generate processing results with low latency.

Another challenge is to handle the state consistency during the prioritized state migration process. In Mecas, the state consistency in the above process is maintained by a control messaging based coordination protocol, inspired by [7, 40, 43]. In addition, as the fetch-on-demand model used in prioritized state migration calls for light-weight state accessing, we devise a hierarchical data organization and adopt a gradual migration strategy for finer-grained state transfer like [26]. Mecas is designed as a pluggable rescaling module without affecting non-rescaling periods. As far as we know, Mecas is the first stateful SPE rescaling approach that enables prioritized state migration, which can reduce the latency spikes without high resource usage or incurring major disruption.

To sum up, this paper makes the following contributions:

- We propose an on-the-fly rescaling mechanism, called Mecas, for stateful distributed stream processing engines. It prioritizes the migration of hot states to achieve low-latency and resource-efficient rescaling.
- In addition, we adopt a control messaging based coordination protocol to maintain state consistency during prioritized state migration. We further devise a hierarchical state data organization and a gradual state migration

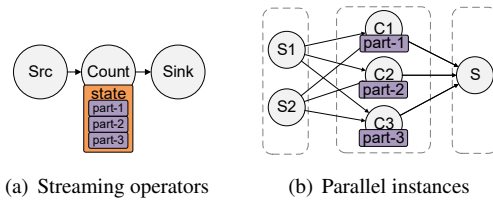


Figure 2: A key-count job example in stream processing

- strategy, which lowers data transmission granularity.
- Mecas is implemented on the widely-used SPE Apache Flink, requiring minimal user code modification and no disruption to non-rescaling periods.
- We validate our design with various workloads. Compared with state-of-the-art approaches, Mecas reduces nearly 95% of processing latency peak during rescaling.

2 Background

In this section, we first introduce basic concepts and terminology of stateful stream processing in Section 2.1, and then review and analyze the design and shortcomings of existing rescaling approaches in Section 2.2.

2.1 Stateful Stream Processing

Stream Processing Topology: In scale-out distributed SPEs, the computation tasks are expressed as directed graphs (Figure 2(a)), where vertices represent stream processing operators. Each operator receives data from its upstream operators, processes data, and sends output to its downstream operators. Operators without upstream are called *source* operators, and those without downstream are called *sink* operators.

SPEs process input data in a data-parallel style by mapping the streaming *operators* to multiple parallel *instances*. The number of an operator’s instances is called its degree of *parallelism*. In Figure 2(b), the vertices represent parallel instances and the directed edges represent data channels.

Stateful Operator: In practice, many analysis tasks require operators to compute their output based on both current and previously received data. Examples of such tasks include data aggregation, window computation, ML models, etc. To realize this, *stateful* operators maintain internal and mutable states. Operator states remember information about past input and can be used for processing of future input [18]. For example, a counting operator of a WordCount job stores the current occurrences of each word as its states.

A stateful operator receives input data as a stream of keyed *records*. Each record can be denoted as a pair (k, v) representing the key and the payload value. Correspondingly, the stateful operator also holds its state S as a set of key-value pairs and divides the set into several disjoint partitions. During

processing, the partition S_l is assigned to an operator instance I . When I receives a record with key k , it processes the record by reading or updating the value v corresponding to k in S_l . In modern SPEs [16, 53], states of the stateful operators are co-partitioned with the parallel instances. For example, in Figure 2(b) the state of the keys mapped to C_1 operator instance is only stored in *part-1*.

Checkpoints: SPEs usually achieve fault-tolerance via checkpoints. A checkpoint marks the persistent state for each operator at a specific time point. SPEs can resume from failure by restoring the operator states with a recent checkpoint and replaying the records since the generation of the checkpoint.

A common way to realize consistent state snapshots of distributed streaming operators is Chandy-Lamport algorithm [10] or its variants. The checkpoint algorithm in Flink [7] works by injecting special streaming records called barriers into the pipeline. Each operator instance will persist its state when receiving a barrier. This workflow makes a globally consistent operator state snapshot on the basis of reliable FIFO data channels.

2.2 Prior Rescaling Mechanisms

The critical difficulty in on-the-fly rescaling distributed stream operators is efficiently migrating states among instances while keeping the exactly-once semantics. Prior approaches can mainly be categorized into the following strategies [6, 26].

Full-Restart: This approach halts the SPE execution, takes a state snapshot of all operators, redistributes the state among operator instances, and restarts the job execution until the state redistribution is complete. Since it is simple and naturally guarantees consistency using existing *checkpoint* mechanisms, it is adopted by many SPEs including Spark Streaming [53], Structured Streaming [5], Apache Flink [16], etc. However, this mechanism halts the whole pipeline and causes serious latency spikes during rescaling.

Partial-Pause: This approach only stalls the streaming operators that need state migration during rescaling. It mitigates the latency spikes by narrowing the interruption from the job-granularity to the operator-granularity. This mechanism is first introduced by Flux [52], and adopted by SEEP [14], FUGU [25], Chi [40], etc. However, if the affected operator is a critical component in the topology, this approach still pauses the entire pipeline during rescaling.

Replicated-Dataflow: This approach replicates the affected operators and executes the old and new configurations simultaneously until the migration completes. It minimizes processing latency and realizes on-the-fly state migration, but requires redundant computing resources for replication [26]. It also calls for additional de-duplication mechanisms as the concurrent execution generates duplicated data. This mechanism is adopted by ChronoStream [61], Gloss [47], etc.

Proactive: Proactive approach adds extra behaviour to non-rescaling periods to relieve the pressure when migrat-

ing states. Megaphone [26] works by splitting operators and embedding the migration flows into data flows. However, it introduces partitioning overhead inside the original processing operators during non-rescaling periods. Besides, in system design it calls for extra coordination and progress tracking mechanisms [26, 43] which are not directly supported by many modern SPEs [16, 53]. Rhino [43] periodically replicates operator states among all workers. It facilitates both fault-tolerance and state migration of extremely large states, but incurs extra network overhead to regular stream processing.

As far as we know, none of the existing approaches consider the state migration problem from the aspect of the state migration order. They rescale with latency spikes or high resource usage [26], or major disruption (see Section 5.6).

3 System Design

In this section, we first introduce the main idea of Meces in Section 3.1. Then, in Section 3.2 we describe the design in detail. Finally, we elaborate the optimization for finer-grained state transfer in Section 3.3 and Section 3.4.

3.1 Prioritized State Migration

If not introducing huge redundant resources [47, 61] or disrupting regular processing [26, 43], prior works [5, 14, 16, 25, 40, 52, 53] fail to achieve low latency when rescaling. They are mainly limited by "order-unaware" state migration. As Figure 1 in Section 1 shows, the migration for hot keys should be prioritized, so as to generate timely results and reduce queuing latency when rescaling.

In order to achieve prioritized state migration, we review the nature of states in streaming operators. We denote a record with key k as r_k , and a state value with key k as v_k . An important fact about recent scale-out stateful SPEs is that the operator states are partitioned across the operator instances in exactly the same way as data records according to their keys. That is to say, when an operator instance I processes a record r_k , the only state that it needs to access is v_k . It is also guaranteed that no common states need to be accessed when processing records with different data keys.

Given this property, we propose the prioritized state migration mechanism, which enables the system rescaling and data processing of SPEs to efficiently run at the same time. Specifically, when the SPE triggers an online state repartition, the previously responsible instances **send** states in batches to the newly responsible instances. In our design, as for the newly responsible instances, instead of only passively waiting for the arrival of migrated state batches, they can also actively **fetch** states from previously responsible instances, to get the individual states corresponding to the data records for timely processing. For example, as no more records with key in $\{k_1, k_2, k_3\}$ will be sent to I_a due to rescaling, the SPE decides to migrate a set of state values $S_m = \{v_{k_1}, v_{k_2}, v_{k_3}\}$

from operator instance I_a to I_b . Then S_m is sent by I_a . Before I_b receives the entire state set, if I_b encounters a record r_{k_2} , it immediately performs a single-value fetch for v_{k_2} . This lightweight operation helps I_b to generate processing results in time, instead of getting blocked until receiving the entire S_m from I_a .

In this way, we can keep the stream processing operators working during SPE rescaling. The batched "send" aims at quick state migration, while the active "fetch" ensures in-time processing for the records that requires a remote state. The processing latency performance will be affected only when an active "fetch" is triggered. In that case, only the processing of that single record is delayed a bit because of one extra state fetch operation, but the queuing cost of subsequent records can be greatly reduced. In other words, the performance interference caused by the state migration comes down to the record-granularity, so that we can keep the stream processing performance as high as possible during the rescaling period.

To achieve this, two obstacles need to be carefully dealt with: First, how to ensure state consistency when transferring states among operator instances in a dynamic order (Section 3.2). Second, how to minimize the performance impact brought by fetch operations (Section 3.3 and 3.4).

3.2 Fetch-on-demand State Accessing Protocol

To support prioritized state migration with dynamic order during rescaling, Mecas leverages a fetch-on-demand state accessing protocol. The state consistency of the fetch-on-demand model is based on a control messaging coordination protocol, inspired by [7, 40, 43]. In the following, we first briefly describe how the protocol works, then use an example to further explain its process.

Migration Process: We call the time period from the beginning to the end of a state redistribution as a *Migration* stage. The global controller of an SPE starts a *Migration* stage by injecting a special data record called *control message* into the source operators. The message then travels through the whole pipeline in the same way as a regular data record. Once an instance I receives a control message from its input data channels, it performs the following steps:

- (1) I sends the control message downstream.
- (2) If the downstream operator needs to migrate states, I updates its routing strategy.
- (3) If I itself needs to migrate states, according to whether it has received messages from all input channels, it successively goes through two phases: *Aligning* and *Aligned*.
- (4) I sends a confirming signal to the global controller. A *Migration* stage ends when the global controller receives confirming signals from all parallel instances.

Figure 3 and Figure 4 show the *Migration* stage during a scale-out operation of a streaming key-count job. The degree of parallelism of the count operator increases from 2 to 3, represented by A, B, C (see Figure 3(a)). The upstream source

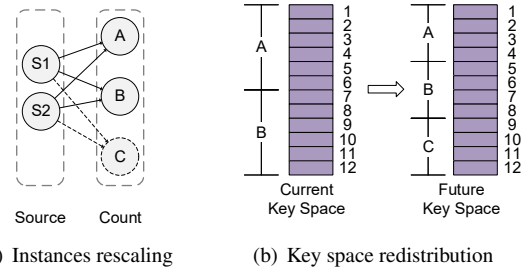


Figure 3: Example of a rescaling stream processing job

operator has two parallel instances $S1, S2$. For simplicity, we assume the keys are between 1 and 12. Therefore, based on the uniform distribution, the distribution of the key space before and after the rescaling is as shown in Figure 3(b).

When $S1$ or $S2$ receives the control messages, it updates the routing strategy and outputs the subsequent data records in accordance with the new topology. As its downstream operator's degree of parallelism will increase by 1, its new strategy divides the key space into three parts equally. As shown in Figure 4(a), the records mapped to "9" were previously sent to B , but will be shipped to the new instance C from now on. Meanwhile, the records with key "6", which were consumed by A , should then be in the charge of B .

As for the count operator, in general, the states are sent from the previously responsible instances, but the newly responsible instances also actively fetch states in response to the incoming data records. We denote an instance's current key space before migration as C_k , and its future key space after migration as F_k . Specifically, the count operator successively goes through two phases. Taking B as an example:

1. **Aligning (Figure 4(b)):** When B first receives a control message, such as from $S2$, it can foresee the arrival of keys that did not belong to it before. After that, when B encounters a record whose key is not in its C_k , such as key "6", instead of considering it as an error, B first checks its F_k . If the key is found, B "borrows" the corresponding state of this key from other count operator instances to complete the processing. Note that in this phase, the message from $S1$ has not reached B , which means A may still have to deal with records with the "6" key. Because of that, B should also be prepared that its state of "6" can still be borrowed back by others.
2. **Aligned (Figure 4(c)):** The *Migration* for B is aligned once it receives control messages from all of its input channels. In this situation, it is guaranteed that all future records with the "5, 6" key are shipped only to B , and B will no longer receive records with keys from 9 to 12. Therefore, B can start its state migration. It checks C_k and F_k , sends the states between 9 and 12 to other instances, and fetches all the states in 5 and 6 asynchronously. When finishing sending and fetching, B sends the completion signal of its *Migration* stage.

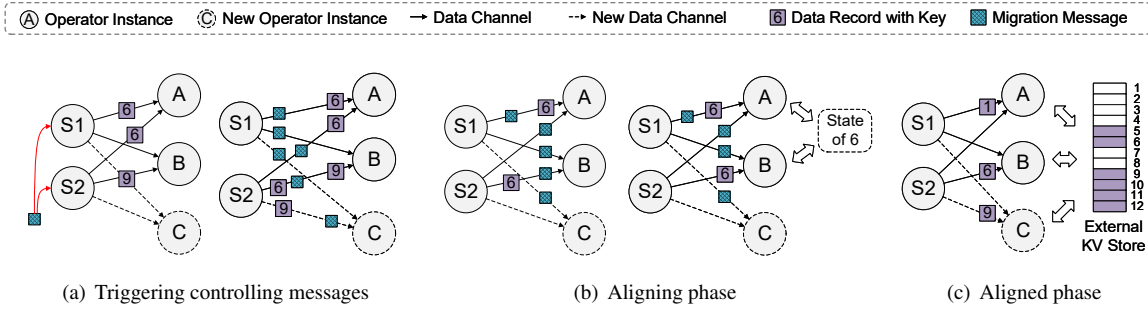


Figure 4: *Migration* stage in a rescaling operation

In the above process, the stream processing operators keep functioning without explicitly blocking the input channels. The impact of state migration on the task is only perceived when an instance requires a remote state. The processing latency of this single record is then only increased by the fetching cost of a single state. For subsequent records, if the corresponding state has already been fetched, the processing of these records suffers from neither migration cost nor huge queuing cost. In this way, the influence of rescaling can be significantly reduced, therefore the system can avoid sudden and severe performance degradation in latency and throughput. For implementation details of the active state fetch process, please refer to Section 4.2.

State Consistency: In stream processing, the global state consistency usually refers to the exactly-once semantics. In Mecas, at a *Migration* stage where the state k is migrated from I_1 to I_2 , each incoming record affects the final results exactly once. Let t_1 be the timestamp when I_1 or I_2 receives the first control message of *Migration*, and t_2 be the timestamp when both of I_1 and I_2 have received control messages from all the input channels. Therefore, the *Aligning* phase begins at t_1 . In this phase, the data records of k can be sent to both I_1 and I_2 , but actually only one instance holds the state of k locally at a time. That is to say, only one instance can modify this state at a time. As the data record must be processed exactly once, and the state can only be flushed and "borrowed" after the processing of the current record is finished, the semantic is kept in $[t_1, t_2]$. After t_2 , subsequent data records are sent to I_2 only. I_2 only needs to borrow at most once to transfer the state to the local and process each data record exactly once until its *Aligned* phase is complete.

3.3 Hierarchical State Data Organization

This subsection proposes an adaptive state data organization, which keeps regular stream running at a coarse granularity to avoid extra overhead, and performs prioritized state migration at a fine granularity to reduce the impact on latency performance of streaming operators.

Since the states in SPEs are key-value data and it is

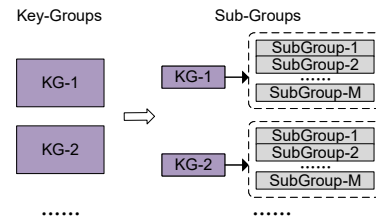


Figure 5: Split key-groups into sub-groups

common to have billions of keys in a real-world streaming dataflow services, managing each key individually can be unrealistic. Many existing SPEs divide the states into key-groups [17], which are disjoint subsets of the entire key space.

However, the shortcoming of this flat index structure is that the state migration is also conducted on the key-group granularity. As the number of the operator states accumulates, the size of one single key-group could grow large. This can make the active "fetch" become time-consuming, bringing long delays to the record processing at the *Migration* stage. A naive solution is to increase the number of key-groups, but this is not practical in many SPEs. A vast number of key-groups will bring a lot of additional metadata management overhead and fragmented read/write of checkpoints, thereby reducing the performance of non-rescaling stream processing.

To address this issue, we introduce the nested layer of state data organization in Mecas. Instead of using a single-layer map, we further divide each key-group into multiple sub-groups as illustrated in Figure 5. When encountering a record that requires a remote state, an instance tries to fetch the corresponding sub-group of the record key instead of the entire key-group. This reduces the time overhead used to obtain data that is not needed immediately.

Note that, key-groups are distributed among different operator instances and the metadata should be stored, indicating which instances are responsible for each key-group. When the number of key-groups increases, it incurs much more metadata management cost and record distribution cost. Differently, sub-groups of the same key-group must belong to the same operator instance when the system is not rescal-

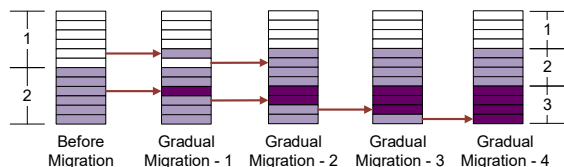


Figure 6: An example of gradual state migration

ing. Therefore, increasing the number of sub-groups does not bring significant extra overhead to a steady dataflow. Users can choose an appropriate number of sub-groups to achieve smooth state migration. The appropriate settings are usually based on the maximum size of states and the expected maximum latency during rescaling.

3.4 Gradual State Migration

During a rescaling operation, a large part of the overall states may need transferring, even if the degree of parallelism does not change much. For example, to divide the states evenly across all operator instances in Figure 3, changing the degree of the parallelism from 2 to 3 causes half of the keys to be redistributed. For example, if we have an operator with 128 key-groups and change the degree of parallelism from 25 to 30, a nearly full state migration (115 out of 128 key-groups) has to be triggered. Migrating all of these states in a single batch can dramatically slow down the overall performance, because most of the records processed by the task in the next period may be affected by the fetch operations. As the data streams continue to flood in, lots of minor processing delays may accumulate into large latency spikes.

To resolve this issue, inspired by [26], we also achieve finer-grained migration via a gradual migration strategy in Mecas, which splits the update into several micro-batches of state migration as shown in Figure 6. The *Migration* stage in one rescaling operation is then composed of multiple *Gradual-Migration* steps. At each step, the global controller decides which states should be relocated based on the user-defined size of micro-batches (*batch_size*). For example, in Figure 6, *batch_size* is set to 1, which means that an instance can only dispose at most one key-group of states at a time. This splits the single *Migration* stage into four *Gradual-Fetch* steps.

At *Gradual-Fetch* steps, the information of migrated keys is included in the control messages and each upstream instance creates a temporary routing table indicating which downstream instance it should send records to. In this way, we affect only a tiny portion of the whole states at each *Gradual-Fetch* step, while most of the records can be processed normally. By changing *batch_size*, users can trade-off the lower latency spikes against higher migration throughput.

Note that during rescaling, the total number of migrated keys can be reduced by dividing the states differently. For example, in Figure 6, if the middle four keys are assigned to instance 3 and the last four keys are assigned to instance 2,

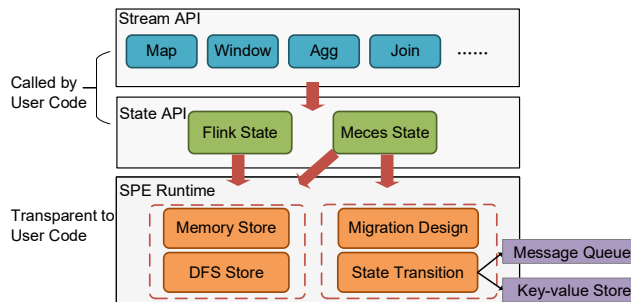


Figure 7: Mecas Architecture

only 4 of 12 keys need relocating. Mecas uses the uniform re-partition by default, but also supports custom partitioning, so that users can apply it to SPEs with sophisticated partitioning approaches [2, 29] such as consistent hashing. A brief evaluation of custom partitioning can be found in Section 5.3.

4 State Migration Implementation

This section describes the implementation of Mecas. We demonstrate the overall system architecture of Mecas in Section 4.1 and then introduce the details of state transition workflow among operator instances in Section 4.2. Finally, we discuss the fault tolerance mechanisms of Mecas in Section 4.3.

4.1 System Architecture and Usage

The overall system architecture of Mecas is shown in Figure 7, which consists of three layers:

1. Stream API: It provides basic operator functions for users to implement stream processing tasks.
2. State API: Mecas provides Apache Flink-compatible APIs for operator functions to access their states. This enables the users to easily migrate the existing stream tasks from Flink to Mecas.
3. SPE Runtime: This is where the system-level code is located. Mecas basically reuses Flink's state backend module to store the key-value pairs in memory/DFS. Additionally, Mecas implements the design in Section 3.

The underlying state management and migration in SPE Runtime are completely transparent to user codes. Therefore, it takes minimal effort to switch between Flink's and Mecas's state implementations.

4.2 State Transition

The state transition among operator instances is of great importance to the performance of SPEs during rescaling. However, if the transfer of states happens directly between parallel instances, a mesh communication network has to be established among the worker machines. This would significantly increase the runtime overhead and make the maintenance of

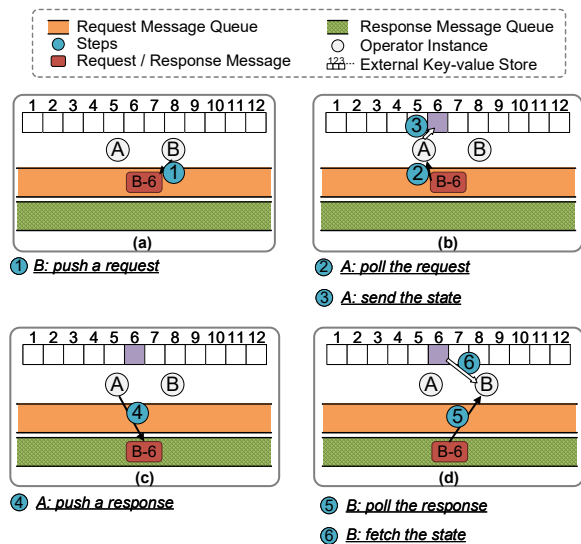


Figure 8: State transition process based on the pub-sub model

SPEs more complex. That is why SPEs like Flink [16] are designed to avoid direct inter-instance communication within an operator, i.e. adopting a shared-nothing architecture [17, 24] for low-cost, high-extensibility and high-availability.

To make it non-disruptive to SPEs, in implementation of Mecas, we design a pub-sub model based approach to transfer states between parallel instances during record processing. The model leverages two message queues (*RequestMQ* and *ResponseMQ*), and an external key-value store (EKS) to provide a state transition service at *Migration* stage. At *Migration* stage, each instance performs light-weighted operations to read from these two message queues continuously.

Figure 8 demonstrates the state transfer process from operator instance *A* to *B*. When *B* needs to fetch the state with key "6", it simply pushes a message "B-6" to the *RequestMQ*, meaning "*B* is requesting 6". Later, *A* gets the message, finds "6" in its C_k , and then pushes the state value into the EKS and considers its local "6" state as borrowed. After that, *A* pushes a message into the *ResponseMQ*, indicating the completion of the request. Eventually, the message will be consumed by *B*, making *B* fetch the corresponding data from EKS. Thus far, the workflow of the state transition is complete.

In the above process, when the request message comes to *A*, if *A* happens to be processing a record with a key of 6, *A* will not start pushing until it finishes processing the current record. Meanwhile, if "6" has already been pushed into EKS and then *A* receives a record requiring this state, *A* should trigger a similar process to fetch the state, as "6" is now considered not held by *A*. These two situations do not bring much degradation to the rescaling performance, because they only happen in the *Aligning* phase, which only lasts for a short time if the system is not running under severe load imbalance.

For decoupling the components, many scalable message queue techniques [27, 46, 49] can be utilized to implement the pub-sub model. These are common components in real-world stream processing tasks, because they suit the stream processing paradigm as data sources and sinks. For EKS, a high-speed key-value store with fault-tolerance guarantees can be integrated to ensure real-time computation.

4.3 Fault Tolerance

Mecas inherits fault-tolerance guarantees from the hosting SPEs, and relies on high-available message-queue/EKS service. Specifically, fault-tolerance is supported in both rescaling and non-rescaling scenarios.

During rescaling: If some of the message-queue/EKS nodes fail, the rescaling in Mecas can keep going without data corruption. This is because message-queue/EKS service adopted by Mecas is equipped with built-in fault-tolerance mechanisms [12, 48], such as replication. If the entire message-queue/EKS or any Flink node fails to respond (simple/cascading failures), Mecas considers this rescaling to have failed and invalidates the temporary data in message-queue/EKS. Then Flink's own failover mechanism is activated to recover the job from a checkpoint. Therefore, a failed rescaling does not break exactly-once semantics.

During non-rescaling: As Mecas is designed to be non-disruptive when rescaling is not executed, it introduces no extra fault-tolerance issues. Any exception is handled by the hosting SPEs' failover mechanism.

Finally, Mecas temporarily disables checkpoint generation only at *Migration* stages, and suspends new rescaling during an ongoing rescaling to avoid interference. Therefore, data consistency is kept throughout the job lifecycle.

5 Evaluation

This section presents an empirical evaluation of our prototype Mecas. We focus on the following three questions:

1. What is the performance of Mecas during rescaling with state migration? (Section 5.2 to 5.4)
2. How much overhead does Mecas incur? (Section 5.5)
3. How does Mecas compare to other state-of-the-art on-the-fly state migration approaches? (Section 5.6)

5.1 Experimental Setup

Hardware and Software: All experiments presented in this paper are conducted on a cluster of six computing nodes, each with two Intel Xeon E5-2620 v2 @2.10 GHz CPUs and 64 GB memory, running CentOS 7.9.2009.

The Mecas prototype is implemented based on Apache Flink 1.12.0. The same version of Apache Flink is used as a baseline in our evaluation, referred to as "Native Flink" in the following. Mecas and Native Flink run with Oracle JVM

11.0.10 to enable the low-latency garbage collector ZGC [63]. We pair Apache Kafka 2.7.0 with Apache ZooKeeper 3.6.0 as reliable message brokers with 25 partitions. Redis 3.2.0 is deployed to provide the external key-value store service globally. For persistent checkpoint storage, we deploy Apache Hadoop 2.9.2 to provide HDFS.

For SPE configurations, we configure Native Flink and Mecas to use at most 25 GB heap memory. Each computing node can host at most 5 parallel instances of an operator. The total number of key-groups is set to 128, as it is the default value in Flink and different settings of this value do not differ the performance much in our cases.

Workloads and Metrics: We choose the NEXMark benchmark suite [55] and the key-count job as workloads. Nexmark models an online auction system and provides real-world streaming queries. The key-count job takes a stream of randomly generated keys as input and accumulates the number of times each key has occurred.

To provide input data for stream processing jobs, we implement open-loop stream generators which continuously and concurrently push random records into Kafka topics. Unless otherwise specified, all these open-loop generators produce data at a steady rate of 800K records/s. This input rate is near the saturation point of processing and large enough to show the performance difference between the various approaches.

We focus on two metrics in measurement during rescaling:

Latency: To evaluate the end-to-end latency of SPEs, we configure the stream generators to periodically insert marker events into Kafka. We denote latency as the time difference between these markers entering and leaving the SPE. For windowed operators, the markers simply bypass them to exclude the time spent in window buffers. The latency still grows when the system’s processing rate cannot keep up with the production speed of upstream data, as the latency markers are queuing up. That is to say, the marker can still reflect the latency performance of the system with windowed operators.

Throughput: We define throughput of SPEs as the number of records output by the source operators per second. As the source operators are responsible for fetching records from Kafka, this reflects the capability of the system to read and process data from external data sources.

5.2 Latency Performance during Rescaling

We first evaluate the latency performance of the SPEs during rescaling. The SPEs are initially configured with global parallelism of 25. Each job runs steadily for 600 seconds and rescales by increasing the parallelism of critical operators (e.g., join or window operators in NEXMark queries, counting operator in the key-count job) to 30. This causes 115 out of 128 key-groups to be relocated during rescaling.

We compare Mecas with Native Flink (stopping the whole job when rescaling) and Order-Unaware (online block-based state migration without order prioritization). Figure 9~12

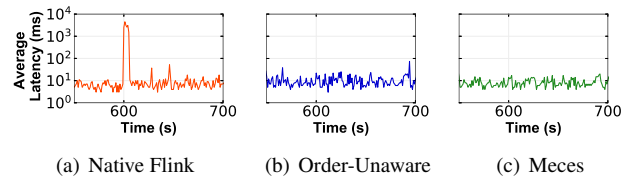


Figure 9: End-to-end latency of NEXMark Q1

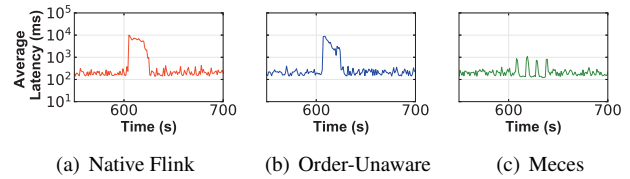


Figure 10: End-to-end latency of NEXMark Q7

illustrate the end-to-end latency change of the representative queries in this process. For the performance evaluation results of all NEXMark queries, please refer to Appendix B.

NEXMark Q1 performs currency conversion on a bid stream. This simple task maintains no states and the behavior is demonstrated as a basic case for our evaluation. As shown in Figure 9, both Mecas and Order-Unaware reveal no latency peak but only system noise, because they incur no state migration cost during rescaling and operations can be done asynchronously. In contrast, Native Flink still needs to stop and restart the job even if there is no state to migrate.

NEXMark Q7 and Q8 tests window operators. They perform tumbling window join of two streams, to find out the items with the highest price and the new users who just registered in the last period of time. They can maintain large states when the window size grows. We set the window sizes to be 10 seconds and 100 seconds. The performance comparison is illustrated in Figure 10 and Figure 11, where the latency peak of Mecas during rescaling is an order of magnitude smaller than others. When the rescaling begins, Order-Unaware needs to block the currently processed records while migrating a considerable amount of states. As a result, Order-Unaware goes through a performance degradation near to Native Flink (Full-Restart), and reaches a latency peak of dozens of seconds. For comparison, Mecas can give state migration priority to the hot keys being processed and smoothen latency.

Key-count takes a stream of randomly generated keys as input and reports the cumulative counts of each key continu-

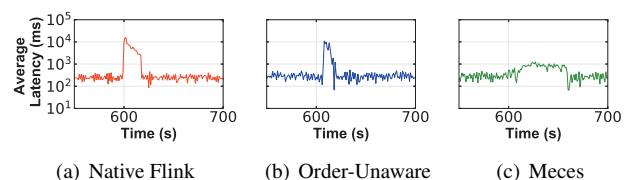


Figure 11: End-to-end latency of NEXMark Q8

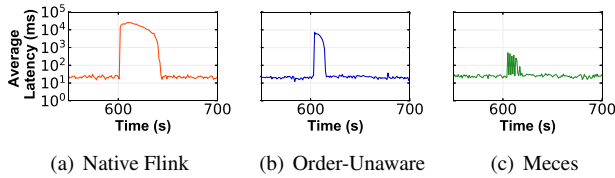


Figure 12: End-to-end latency of key-count

ously. The state size of the counting operator can also grow large when the key range is big enough. Besides, this query requires reading and updating the states when processing every single record. We run the job with 10^8 unique keys. As reported in Figure 12, both Native Flink and Order-Unaware show a latency peak which is three orders of magnitude higher than usual. The latency decreases gradually after the restarting or migration completes. As for Mecas, it keeps the latency under 600 ms during the prioritized state migration stages.

In conclusion, during rescaling, Native Flink and SPEs with Order-Unaware can suddenly become out-of-service when migrating states, while Mecas significantly lowers the maximum latency. The impact of prioritized migration of Mecas is further evaluated in Section 5.3. Note that in some cases with a rather large size of states, Mecas also have increased latency. This is mainly caused by the Garbage Collection behaviour of JVM, which is further analyzed in Section 5.5.

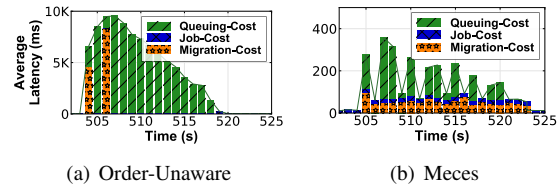
5.3 Impact of State Migration on Latency Performance during Rescaling

In this subsection, we evaluate how state migration affects the processing latency of SPEs, especially how the prioritized migration improves the performance.

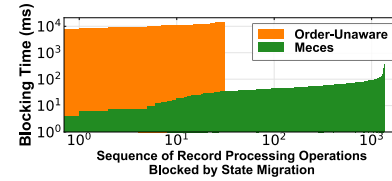
During a rescaling period involving state migration, the overall processing latency of SPEs can be generally divided into three parts: (1) Job-Cost: time to execute the processing logic. This is inevitable in both rescaling and non-rescaling periods; (2) Migration-Cost: when encountering a record whose state is not local, the task waits for the target states to arrive and then proceeds processing; (3) Queuing-Cost: If a record r is blocked due to migration cost, subsequent records may also be blocked in the queue as they cannot be processed until the processing of r is finished. Therefore the processing latency of these subsequent records is increased.

Figure 13 illustrates the various parts of the average processing latency of a certain operator instance per second during the rescaling of the key-count job. Note that in order to show the comparison between different parts, here we use linear axes instead of logarithmic axes and draw with different y-axis ranges in the upper two sub-figures.

As in Figure 13(a), Order-Unaware incurs high latency up to thousands of milliseconds, and the increased latency is composed of huge Migration-Cost and Queuing-Cost. Because Order-Unaware does not migrate the currently needed states



(a) Order-Unaware (b) Mecas



(c) Distribution of Migration-Cost for all operator instances

Figure 13: Latency composition of during rescaling

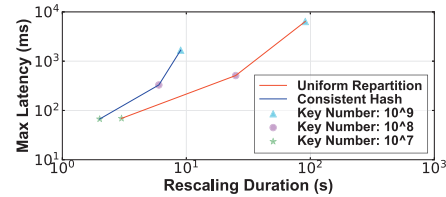


Figure 14: Rescaling performance of Mecas using different repartition strategies

with priority, some of records have to wait for a long time before the arrival of their corresponding states. The waiting time can be up to several seconds as indicated by the Migration-Cost bar in the figure. As for the subsequent records, their states have been previously transferred and they need not wait for data migration. However, they suffer from huge Queuing-Cost due to the previous records being blocked. When no further migration is required, the Queuing-Cost decreases to 0 almost linearly.

Figure 13(b) reports the latency breakdown of Mecas. Compared with Order-Unaware, the latency peak is greatly reduced to less than 400 ms. This is because when migrating states, Mecas uses its fetch-on-demand model to give priority to the records which are currently being processed. When a record calls for a remote state, the operator instance immediately fetches the single state for the key of this record. Since the amount of data in a single state is very small, a fetch request can be quickly responded to. Consequently, as shown in Figure 13(c), the several long-duration execution blocks caused by Migration-Cost are converted into thousands of short-duration fetch operations. Although the sum of Migration-Cost does not differ much for two strategies, each record does not wait long time for its state to be migrated in the prioritized migration of Mecas. More importantly, as the Migration-Cost of every single record is reduced, the Queuing-Cost for subsequent records is also significantly reduced. Eventually, the overall latency curve is flattened.

From another angle, Figure 14 compares the rescaling per-

formance of Mecas using different state repartition strategies: the default uniform repartition and consistent hashing, which have different migration cost during rescaling. For a total of 128 key-groups, consistent hashing decreases the number of migrated key-groups from 115 to 15. Eventually up to 70% rescaling duration and 90% max latency are reduced. This indicates that we can equip Mecas with existing re-balance technology to improve the rescaling performance. For a fair comparison, in all other experiments in this paper, we use the same repartition strategy in all SPEs including Mecas.

Note that the above experiments are all conducted without any node failure or connection loss. As for the unusual scenarios including failing nodes with different roles, we observe that: (1) If any Flink node or Kafka/Redis leader fails, the latency curve is similar to Native Flink, because Mecas restarts the job since the underlying service becomes unresponsive. (2) If Kafka/Redis loses some of the follower nodes but still provides timely service when job is rescaling, there is no observable fluctuation in the average latency of fetch operations, because of the relatively low traffic of messages sent by fetch operations. In both cases there is no data inconsistency in the stream processing results.

5.4 Performance under Backpressure

In practice, the rescaling of an SPE is usually triggered when it sends a backpressure signal, indicating that it cannot process data fast enough to keep up with the data generation rate. This happens when there is a sudden surge in data traffic or when the system is not configured properly with enough parallel instances. To validate our design in this situation, we evaluate the performance of Mecas under backpressure scenarios.

A costly version of key-count query is chosen as the workload. We first configure the counter operator parallelism to 15 and run input generators at a speed of 300K records/s for 600 seconds. After that the input rate is increased to 600K records/s for 150 seconds, and then gets back to 300K records/s. This simulates a temporary surge in data traffic. The rescaling operation takes place at the 620th second, which increases the counter operator to 30 parallel instances.

Figure 15(a) shows how quickly the SPE recovers its real-time performance from backpressure. As soon as the data traffic surge comes, the latency increases suddenly, because the input is beyond the capability of the system and the following records are queuing up. Then, after the scale-out operation is completed and all the queuing records have been fully consumed, the processing latency can go back to a normal low level. Here we record the time interval from the arrival of the data surge until the system latency drops below 100 ms. As can be seen, Mecas is the first one to get back to the previous processing rate, while it takes Native Flink and Order-Unaware much more time to consume the queuing data and recover because of the block of operator execution.

We then compare the system throughput within 2 min-

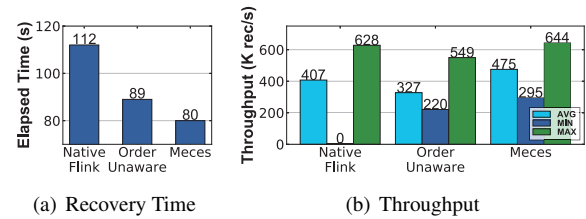


Figure 15: Rescaling performance under backpressure

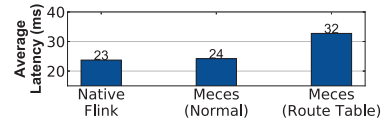


Figure 16: Latency comparison with Native Flink

utes after triggering the rescaling operation, as shown in Figure 15(b). As Native Flink first triggers a global state snapshot and then restarts the job, its throughput immediately decreases to 0 and gradually increases after it restarts. As for Mecas and Order-Unaware, they both keep the throughput at a non-zero level, but as Order-Unaware incurs long-duration blocking periods, it first goes through a decrease in throughput. Meanwhile, as Mecas brings little degradation to the performance during the state migration, its throughput reaches the maximum much faster and higher than the others. This validates that the fetch-on-demand state migration mechanism of Mecas outperforms the other approaches under backpressure.

5.5 Overhead Analysis

This subsection discusses the overhead introduced by Mecas.

Latency Overhead: The latency performance is illustrated in Figure 16. On one hand, Mecas does not incur extra latency when not migrating states, as reported in the "Mecas (Normal)" bar. Under normal circumstances, the processing logic of Mecas and Native Flink is substantially the same. The only difference is that Native Flink uses a normal HashMap to manage operator states, while Mecas uses a nested HashMap with little overhead. Consequently, the difference in latency between the two SPEs is determined by system noise. This conclusion can also be drawn from Figure 9~12.

On the other hand, the processing latency increases when Mecas is performing a rescaling operation. In this experiment, each route table contains 128 keys and increases the latency by 35%. During these periods, when deciding which downstream operator to send records to, the simple modulo operation is replaced by a more costly map query operation. In Mecas, this mechanism is combined with the nested data structure, to reduce the migration granularity to an acceptable level without the need for an extremely high number of key groups. That means the route tables are kept in a reasonably small size, thus reducing the latency peak with negligible overhead.

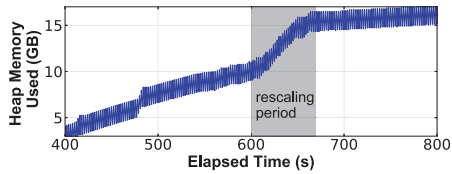


Figure 17: Memory consumption of Mecos

Memory Overhead: Figure 17 demonstrates the memory consumption of one single machine. To clearly show the changes in memory usage, we configure the JVM to utilize `g1gc` as the garbage collector and do not trigger GC behavior for objects in old generation. The memory usage fluctuates up and down because JVM periodically reclaims the temporary objects in the continuously arriving data stream. In the meantime, the total memory usage shows an upward trend due to the gradual increase in the operator states.

When the state migration starts at 600 s, the curve rises rapidly, because the operator starts fetching states from others, allocating lots of new objects quickly. This can potentially decrease the processing performance if the system is implemented in programming languages like Java, as the garbage collector works under heavy pressure and can block the execution of user functions. To ensure the quick response of the system, we recommend a low-latency collector like ZGC [63], or a pre-allocated object pool to be utilized to reduce the overhead of Garbage Collection behavior.

5.6 Comparison with Other State Migration Approaches

In this subsection, we compare Mecos with two representative state-of-the-art work Rhino [43] and Megaphone [26]. We choose the key-count job due to its simplicity so that we can minimize interference from associated computation and highlight the differences between rescaling approaches.

5.6.1 Comparison with Rhino

Rhino [43] proposes a state management approach which handles large states very well by periodically replicating operator states among all worker machines. Because the source code of Rhino from the original authors is not publicly available, we implement the mechanism of Rhino based on Flink for a fair comparison. During a stateful rescaling, Rhino generally follows the Partial-Pause approach, but only reads/writes the incremental parts of states since the last global replication. The replication interval is set to 60 seconds.

As in Figure 18, Rhino on Flink shows a similar latency peak to Order-Unaware, whose peak is near 10,000 ms. As a comparison, the per-record latency of Mecos never reaches 1000 ms during the whole process. Rhino’s replication fails to improve the system performance for two reasons:

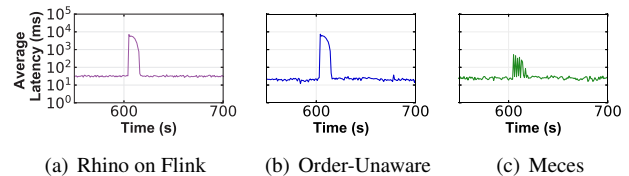


Figure 18: Latency comparison with Rhino during rescaling

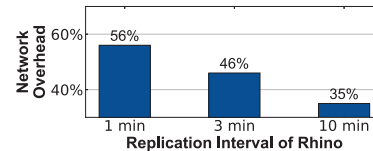


Figure 19: Network overhead of Rhino

1. In such a scale-out scenario where new workers join the job, a global state migration is still necessary for these new workers because they lack the previous states.
2. Such read-modify-write jobs update the operator states very frequently. As lots of states are modified between two state replications, the incremental parts still occupy large proportions of the global state.

Eventually Rhino degrades to the Partial-Pause approach with latency spikes. In contrast, Mecos uses a fetch-on-demand state accessing model for efficient state sharing among operator instances and less processing latency.

In addition, we compare Rhino and Mecos in terms of the disruption to non-rescaling stream processing, by measuring the extra network bandwidth introduced by both systems compared to Native Flink. In Rhino, when replication interval decreases from 10 minutes to 1 minute, the extra network bandwidth ratio grows from 35% to 56% (Figure 19). This is consistent with the conclusion reported in the Rhino paper [43], where Rhino uses 30% network bandwidth during a replication. As Rhino periodically replicates operator states among all workers, it helps fault-tolerance in the face of very large states, but incurs extra communication across the network even if the system performs no state migration at all. In contrast, Mecos incurs no network overhead to non-rescaling stream processing, because it performs no additional operations during non-rescaling periods.

5.6.2 Comparison with Megaphone

Megaphone [26] is a state migration approach that splits the state load into batches and embeds the migration flows into data flows for lower latency. It relies on two specific SPE features [26]: (1) state extraction from upstream operators (2) dataflow frontiers. However, both are still not natively supported in many widely-used SPEs, including Flink [16], Heron [33], Spark Streaming [53], Samza [50], etc. Therefore, to run Megaphone in a widely-used SPE, we meet the above requirements of Megaphone in Flink with naive implementations for state extraction and splitting the data stream

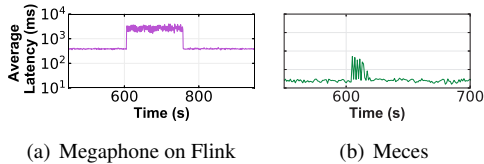


Figure 20: Latency comparison with Megaphone

into micro batches underhook. Based on that, we implement Megaphone’s state migration mechanism on Flink.

We then run the key-count job in both Mecas and Megaphone with 10^8 unique keys. Megaphone is configured to use 2^{14} bins, according to the original suggestions [26]. The latency comparison is demonstrated in Figure 20.

When not rescaling, Megaphone on Flink shows an order of magnitude higher latency than Mecas. There are two reasons for this phenomenon: (1) Megaphone’s strong pre-requisite of SPE features calls for extra synchronization and communication techniques to fulfill the requirements, thus bringing dramatic overhead to the system performance [43]. (2) To prepare for state migration, Megaphone splits the original operators into two operators before the streaming job is submitted. However, this incurs extra partition overhead between the two new operators, resulting in increased processing latency for the regular execution of the job. As for Mecas, it can work without expensive system requirements or logic modification to a non-rescaling SPE, and thus brings no overhead when the system is not rescaling.

During rescaling, both systems show a limited amount of latency increase. However, the latency of Megaphone stays at the level of around 8000 ms, while Mecas never reaches the bar of 1000 ms. This verifies that compared with Megaphone, Mecas’s prioritized state migration can efficiently reduce the processing latency during rescaling with low overhead.

6 Related Work

Stateful stream processing has been an active research field in the past years, both in single-machine [1, 9, 44] and distributed settings [3, 8, 16, 50, 53, 62]. To meet diversified requirements of real-time computing in different scenarios, research works focus on various aspects of stream processing, including performance [32, 36, 37, 58, 59], reliability [45, 64], scalability [14, 15, 56], flexibility [23], programmability [5, 40], etc. In addition, some researchers propose SPEs with enriched semantics [38, 42, 57, 65, 66] to support more sophisticated analysis of streaming data.

Elasticity in batch processing systems has also been studied [34, 35], but they mainly deal with scenarios with rather higher processing latency. As for the field of elastic stream processing, the past decade also witnessed many advances [14–16, 25, 26, 33, 40, 43, 47, 52, 53, 61].

The most recent works related to ours are Megaphone [26] and Rhino [43]. Megaphone [26] provides efficient on-the-fly

state migration for SPEs. This is achieved by transferring the operator states in a small granularity upon dataflow reconfiguration. It also supports trading off low latency against high throughput of state migration. Rhino [43] periodically replicates operator states among all worker machines. It can speed up the process of state migration by asking the operator instances to read/write from an incremental checkpoint instead of a global state snapshot. They both reduce processing latency during state migration at the expense of SPE performance during non-rescaling periods, as neither of them considers the migration order of states. In contrast, Mecas uses a fetch-on-demand state accessing mechanism to enable prioritized state migration during rescaling, without extra resource usage in non-rescaling periods.

Another critical issue about the elasticity of stream processing is to decide when and how to rescale. Many SPE controllers [4, 19, 20, 28, 31, 39, 41, 54] have been proposed for adaptive rescaling to meet the QoS targets in various scenarios. These works are orthogonal to ours and can be combined with Mecas’s on-the-fly rescaling mechanism to provide self-regulating streaming systems.

7 Conclusion

This paper presents Mecas, a latency-efficient on-the-fly rescaling mechanism using prioritized state migration for stateful distributed SPEs. Mecas uses a fetch-on-demand model with hierarchical state data structure and gradual strategy, to achieve prioritized state migration with global consistency and high efficiency. This design puts all the operations in rescaling periods and requires no work during non-rescaling periods. We implement Mecas in Apache Flink and evaluate our design on diversified workloads. The experimental results show that compared to state-of-the-art approaches, Mecas improves the latency and throughput performance during rescaling by orders of magnitude without disrupting non-rescaling periods or using huge amounts of resources.

In the future, we plan to integrate Mecas with stream performance monitoring tools and further study more adaptive rescaling policies for diversified scenarios on Mecas.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. We sincerely thank our shepherd for her/his guidance and time. This work is funded in part by the National Natural Science Foundation of China (No. 62072230), Alibaba Group through Alibaba Innovative Research Program, the National Natural Science Foundation of China (No. U1811461), Jiangsu Province Science and Technology Key Program (No. BE2021729), the Collaborative Innovation Center of Novel Software Technology and Industrialization. Rong Gu is the corresponding author of this paper.

References

- [1] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal '03*, 12(2):120–139, November 2003.
- [2] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 739–753, November 2016.
- [3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In *Proceedings of the VLDB Endowment (PVLDB '15)*, volume 8, pages 1792–1803, April 2015.
- [4] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. Adaptive control of extreme-scale stream processing systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS '06)*, pages 71–71, July 2006.
- [5] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative API for real-time applications in Apache Spark. In *Proceedings of the 37th ACM International Conference on Management of Data (SIGMOD '18)*, pages 601–613, June 2018.
- [6] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. Beyond analytics: The evolution of stream processing systems. In *Proceedings of the 39th ACM International Conference on Management of Data (SIGMOD '20)*, pages 2651–2658, June 2020.
- [7] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*, 2015.
- [8] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. In *Proceedings of the 31st ACM Conference on Programming Language Design and Implementation (PLDI '10)*, pages 363–375, June 2010.
- [9] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. In *Proceedings of the VLDB Endowment (PVLDB '14)*, volume 8, pages 401–412, April 2014.
- [10] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS '85)*, 3(1):63–75, March 1985.
- [11] Debezium. Building audit logs with change data capture and stream processing. <https://debezium.io/blog/2019/10/01/audit-logs-with-change-data-capture-and-stream-processing/>, 2019.
- [12] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations. In *Proceedings of the 11st ACM International Conference on Distributed and Event-based Systems (DEBS '17)*, pages 227–238. ACM, June 2017.
- [13] Exastax. Real-time stream processing for internet of things. <https://medium.com/@exastax/real-time-stream-processing-for-internet-of-things-24ac529f75a3/>, 2017.
- [14] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 32nd ACM International Conference on Management of Data (SIGMOD '13)*, pages 725–736, June 2013.
- [15] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. Making state explicit for imperative big data processing. In *Proceedings of 2014 USENIX Annual Technical Conference (ATC '14)*, pages 49–60, June 2014.
- [16] Apache Flink. <https://flink.apache.org/>, 2015.
- [17] A deep dive into rescalable state in Apache Flink. <https://flink.apache.org/features/2017/07/04/flink-rescalable-state.html/>.
- [18] flink-rescalable-state. <https://flink.apache.org/features/2017/07/04/flink-rescalable-state.html/>, 2021.
- [19] Avrielia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: Self-regulating stream processing in heron. In *Proceedings of the VLDB*

Endowment (PVLDB '17), volume 10, pages 1825–1836, April 2017.

- [20] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. DRS: Auto-Scaling for real-time stream analytics. *IEEE/ACM Transactions on Networking (TON '17)*, 25(6):3338–3352, June 2017.
- [21] Yupeng Fu and Chinmay Soman. Real-time data infrastructure at uber. In *Proceedings of the 40th ACM International Conference on Management of Data (SIGMOD '21)*, pages 2503–2516, June 2021.
- [22] Can Gencer, Marko Topolnik, Viliam Durina, Emin Demirci, Ensar B. Kahveci, Ali Gürbüz, József Bartók, Grzegorz Gierlach, Frantisek Hartman, Ufuk Yilmaz, Ondrej Lukás, Mehmet Dogan, Mohamed Mandouh, Marios Fragkoulis, and Asterios Katsifodimos. Hazelcast jet: Low-latency stream processing at the 99.99th percentile. In *Proceedings of the VLDB Endowment (PVLDB '21)*, volume 14, pages 3110–3121, 2021.
- [23] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Tappan Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *Proceedings of the 13rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pages 213–231, October 2018.
- [24] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems (TPDS '12)*, 23(12):2351–2365, 2012.
- [25] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*, pages 13–22, May 2014.
- [26] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. Megaphone: Latency-conscious state migration for distributed streaming dataflows. In *Proceedings of the VLDB Endowment (PVLDB '19)*, volume 12, pages 1002–1015, April 2019.
- [27] Apache Kafka. <http://kafka.apache.org/>, 2011.
- [28] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava C. Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *Proceedings of the 13rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pages 783–798, October 2018.
- [29] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th ACM Symposium on the Theory of Computing (STOC '97)*, pages 654–663, May 1997.
- [30] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream data processing systems. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE '18)*, pages 1507–1518, August 2018.
- [31] Alireza Khoshkbarforousha, Alireza Khosravian, and Rajiv Ranjan. Elasticity management of streaming data analytics flows on clouds. *Journal of Computer and System Sciences (JCSS '17)*, 89:24–40, October 2017.
- [32] Alexandros Koliouisis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter R. Pietzuch. SABER: Window-Based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 35th ACM International Conference on Management of Data (SIGMOD '16)*, pages 555–569, July 2016.
- [33] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 34th ACM International Conference on Management of Data (SIGMOD '15)*, pages 239–250, May 2015.
- [34] Alok Kumbhare, Marc Frincu, Yogesh Simmhan, and Viktor K. Prasanna. Fault-tolerant and elastic streaming mapreduce with decentralized coordination. In *Proceedings of the 35th International Conference on Distributed Computing Systems*, pages 328–338, June 2015.
- [35] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. A study of skew in mapreduce applications. In *Proceedings the 5th Open Cirrus Summit*, pages 1–5, June 2011.
- [36] Gyewon Lee, Jeongyoon Eo, Jangho Seo, Taegeon Um, and Byung-Gon Chun. High-performance stateful stream processing on solid-state drives. In *Proceedings of the 9th ACM Asia-Pacific Workshop on Systems (APSys '18)*, pages 9:1–9:7, August 2018.

- [37] Yongkun Li, Zhen Liu, Patrick PC Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. Differentiated Key-Value storage management for balanced I/O performance. In *Proceedings of 2021 USENIX Annual Technical Conference (ATC '21)*, pages 673–687, July 2021.
- [38] Thomas Lindemann, Jonas Kauke, and Jens Teubner. Efficient stream processing of scientific data. In *Proceedings of the 34th IEEE International Conference on Data Engineering Workshops (ICDEW '18)*, pages 140–145, April 2018.
- [39] Björn Lohrmann, Peter Janacik, and Odej Kao. Elastic stream processing with latency guarantees. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems (ICDCS '15)*, pages 399–410, June 2015.
- [40] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanam Muthukrishnan, Vamsi Kuppala, Sudheer Dhulipalla, and Sriram Rao. Chi: A scalable and programmable control plane for distributed stream processing systems. In *Proceedings of the VLDB Endowment (PVLDB '18)*, volume 11, pages 1303–1316, April 2018.
- [41] Tiziano De Matteis and Gabriele Mencagli. Elastic scaling for distributed latency-sensitive data stream operators. In *Proceedings of the 25th IEEE EuroMicro International Conference on Parallel, Distributed and Network-based Processing (PDP '17)*, pages 61–68, March 2017.
- [42] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. In *Proceedings of the VLDB Endowment (PVLDB '20)*, volume 13, pages 1793–1806, March 2020.
- [43] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Rhino: Efficient management of very large distributed state for stream processing engines. In *Proceedings of the 39th ACM International Conference on Management of Data (SIGMOD '20)*, pages 2471–2486, June 2020.
- [44] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 439–455, November 2013.
- [45] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*, pages 1–14, April 2013.
- [46] Rabbitmq. <https://www.rabbitmq.com/>, 2007.
- [47] Sumanaruban Rajadurai, Jeffrey Bosboom, Weng-Fai Wong, and Saman P. Amarasinghe. Gloss: Seamless live reconfiguration and reoptimization of stream programs. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*, pages 98–112, March 2018.
- [48] Redis Sentinel. <https://redis.io/topics/sentinel/>, 2021.
- [49] Apache RocketMQ. <http://rocketmq.apache.org/>, 2012.
- [50] Apache samza. <http://samza.apache.org/>, 2013.
- [51] Twitter sentiment analysis: A tale of stream processing. <https://towardsdatascience.com/twitter-sentiment-analysis-a-tale-of-stream-processing-8fd92e19a6e6/>, 2020.
- [52] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE '03)*, pages 25–36, March 2003.
- [53] Apache Spark Streaming. <https://spark.apache.org/streaming/>, 2013.
- [54] Rafael Tolosana-Calasanz, Javier Diaz Montes, Omer F. Rana, and Manish Parashar. Feedback-control & queuing theory-based resource management for streaming applications. *IEEE Transactions on Parallel and Distributed Systems (TPDS '17)*, 28(4):1061–1075, October 2017.
- [55] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. Nexmark - a benchmark for queries over data streams draft. Technical report, School of Science and Engineering at OHSU, March 2008.
- [56] Taegeon Um, Gyewon Lee, Sanha Lee, Kyungtae Kim, and Byung-Gon Chun. Scaling up IoT stream processing. In *Proceedings of the 8th ACM Asia-Pacific Workshop on Systems (APSys '17)*, pages 25:1–25:7, September 2017.
- [57] Pourya Vaziri and Keval Vora. Controlling memory footprint of stateful streaming graph processing. In *Proceedings of 2021 USENIX Annual Technical Conference (ATC '21)*, pages 269–283, July 2021.

- [58] Uri Verner, Assaf Schuster, and Mark Silberstein. Processing data streams with hard real-time constraints on heterogeneous systems. In *Proceedings of the 25th ACM International Conference on Supercomputing (ICS '11)*, pages 120–129, May 2011.
- [59] Uri Verner, Assaf Schuster, Mark Silberstein, and Avi Mendelson. Scheduling processing of real-time data streams on heterogeneous multi-gpu systems. In *Proceedings of the 5th ACM International Systems and Storage Conference (SYSTOR '12)*, pages 1–12, June 2012.
- [60] Virtuslab. Preventing fraud and fighting account takeovers with kafka streams. <https://www.confluent.io/blog/fraud-prevention-and-threat-detection-with-kafka-streams/>, 2020.
- [61] Yingjun Wu and Kian-Lee Tan. Chronostream: Elastic stateful stream computation in the cloud. In *Proceedings of the 31st IEEE International Conference on Data Engineering (ICDE '15)*, pages 723–734, April 2015.
- [62] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 423–438, November 2013.
- [63] The Z Garbage Collector. <https://wiki.openjdk.java.net/display/zgc/Main/>, 2021.
- [64] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, pages 1187–1204, November 2020.
- [65] Yunhao Zhang, Rong Chen, and Haibo Chen. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, pages 614–630, November 2017.
- [66] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proceedings of the 37th ACM International Conference on Management of Data (SIGMOD '18)*, pages 741–756, June 2018.

A Artifact Appendix

Abstract

Meces is implemented on Apache Flink. It also relies on Kafka and Redis to function properly. We prepare the programs, assemble a workflow of Meces and package the artifact into a Docker image.

Scope

The artifact rescales a key-count job with different state migration mechanisms. It verifies the basic functions of Meces and validates the performance improvement brought by the prioritized state migration strategy proposed in this paper.

Contents

The artifact includes a compiled version of Meces, along with dependencies such as Kafka, Redis, Java and Python, etc. A "README.md" file can be also found in the image. It contains detailed description of the artifact and a step-by-step instruction for the rescaling workflow.

Hosting

The artifact is hosted on Docker Hub. It can be installed by downloading the pre-built Docker image from the public dockerhub repository and initiating a container from it:

- `docker pull njupalab/meces:latest`
- `docker run -it njupalab/meces:latest`

Artifact Check-list:

- Run-time Environment: Linux OS with Docker installed.
- Experiments: Workflow of rescaling a key-count job.
- Expected Experiment Running Time: About half an hour.
- Output: Data plots of latency comparison.

Expected Running Result

There are scripts that help pre-check the testing environment:

- `source scripts/install.sh`
- `scripts/start_background_environment.sh`
- `scripts/check_environment.sh`

Then, an experiment script evaluates the rescaling performance of Meces using the key-count job on your environment:

- `scripts/rescale_exp.sh`

It generally goes through three stages in series, namely *Meces*, *Order-Unaware*, *Native-Flink*. In each stage, the system submits the key-count job, runs for a while and then rescales the operator with its corresponding mechanism. For further details, please refer to "README.md" in the artifact.

After the process finishes, the experimental data is collected in the *data* folder. Each experiment should generate a plot of the latency curve, similar to what is reported in Section 5.2.

B Appendix: Latency Performance Evaluation on full NEXMark suite during Rescaling

We evaluate the latency performance of the SPEs during rescaling. The SPEs are initially configured with global operator parallelism of 25. Then, each job runs at a steady input rate of 800K records/s for 600 seconds. After that a rescaling operation is triggered. It increases the parallelism of critical operators (e.g., join or window operators in NEXMark queries, counting operator in the key-count job) to 30. This causes 115 out of 128 key-groups to be relocated during the scale-out.

Figure 21~28 illustrate the end-to-end latency change of each query in this process. We compare Mecas with Native Flink (stopping the whole job when rescaling) and Order-Unaware (online block-based state migration without order prioritization).

NEXMark Q1 and Q2 do currency conversion or filtering operations on a bid stream. These simple transformation tasks do not maintain states and the behavior is demonstrated as a basic case for our evaluation. As shown in Figure 21 and 22, both Mecas and Order-Unaware reveal no latency peak but only system noise, because they incur no state migration cost during rescaling and operations can be done asynchronously. In contrast, Native Flink still needs to stop and restart the job even if there is no state to migrate.

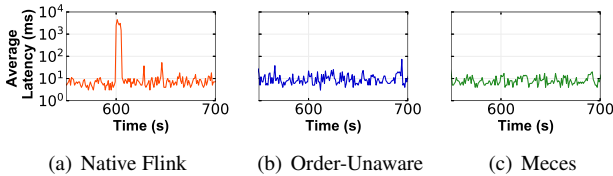


Figure 21: End-to-end latency of NEXMark Q1

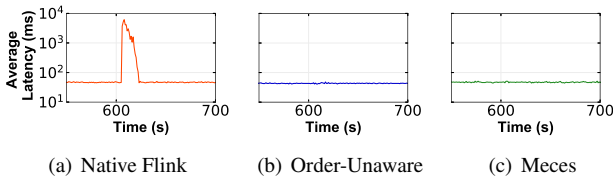


Figure 22: End-to-end latency of NEXMark Q2

NEXMark Q3 and Q4 test join functionality. Q3 joins the stream of open auctions and the stream of people to local item suggestions for users. Q4 joins the stream of closed auctions and the stream of items to output the average deal price of items for a category. Both queries have to store information of records in the data stream as states of join operators. The per-record latency is demonstrated in Figure 23 and Figure 24. In our settings, Q3 maintains a rather small size of states. Meanwhile, operator states in Q4 grow large when millions of items have been sold. As a result, for Native Flink and

Order-Unaware, a sharp and short-lasting rise of latency can be seen in Figure 23, because the task executions are globally or partially blocked until the state migration is completely done. As a comparison, there is no obvious latency change in Mecas as it reduces the disturbance caused by state migration to a lower granularity. A similar conclusion can also be drawn from the results in Figure 24 for Q4. The degradation of real-time performance becomes more significant for Native Flink and Order-Unaware with a larger size of states, while Mecas only incurs a small range of latency fluctuations. The latency peak of Mecas is an order of magnitude smaller than the other two mechanisms, due to its fetch-on-demand accessing and gradual migration strategy.

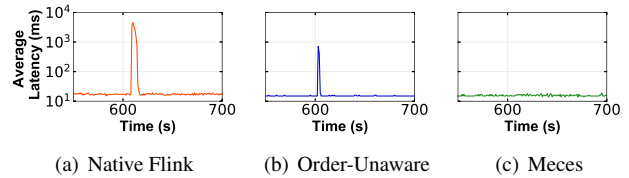


Figure 23: End-to-end latency of NEXMark Q3

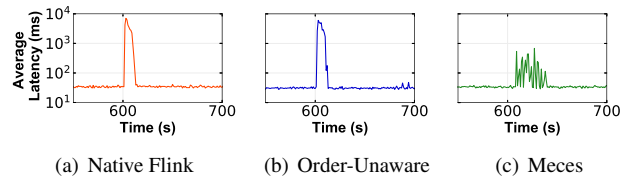


Figure 24: End-to-end latency of NEXMark Q4

NEXMark Q5 tests window operator with small states. It repeatedly selects the hottest item in the past period of time, which is the item with the most number of bids. The stateful operator maintains item counts in each time window. In our experiments, the job reports every second the hottest item over the last 60 seconds. As this query does not accumulate large states, it exposes similar behavior with Q1 and Q2, as shown in Figure 25. While the latency of Native Flink increases significantly due to a full restart, the record processing of Mecas and Order-Unaware is hardly affected because the state migration cost is minor.

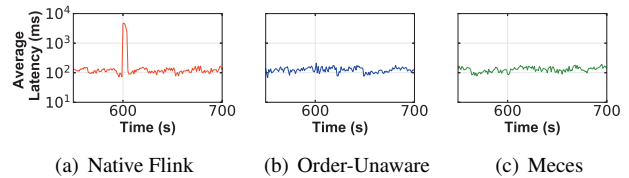


Figure 25: End-to-end latency of NEXMark Q5

NEXMark Q6, Q7 and Q8 test window operator with bigger states. Q6 includes a sliding window over a single

stream to calculate the average of items recently sold by a seller. Q7 and Q8 perform tumbling window join of two streams, to find out the items with the highest price and the new users who just registered in the last period of time. These queries maintain large states when the window size grows. We set the window sizes to be 10 seconds, 10 seconds, and 100 seconds for the three queries respectively. The performance comparison is illustrated in Figure 26~28, where the latency peak of Mecas during rescaling is an order of magnitude smaller than the others. When the rescaling begins, Order-Unaware needs to block the currently processed records while migrating a considerable amount of states. As a result, Order-Unaware goes through a performance degradation near to Native Flink (Full-Restart), and reaches a latency peak of dozens of seconds. For comparison, Mecas can give priority to the hot keys being processed and smoothen the latency peaks.

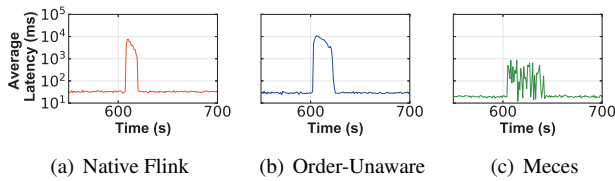


Figure 26: End-to-end latency of NEXMark Q6

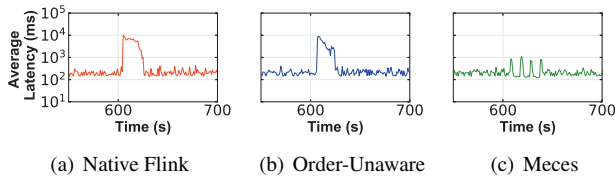


Figure 27: End-to-end latency of NEXMark Q7

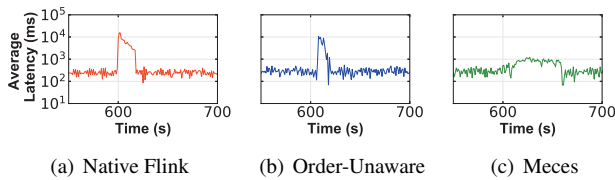


Figure 28: End-to-end latency of NEXMark Q8

In conclusion, Mecas achieves better performance than existing approaches during rescaling. Native Flink and SPEs with Order-Unaware can suddenly become out-of-service when migrating states, while Mecas relieves the impact of rescaling to a lower granularity and reduces the maximum latency.



DepFast: Orchestrating Code of Quorum Systems

Xuhao Luo[†], Weihai Shen[‡], Shuai Mu[‡], Tianyin Xu[†]

[†]University of Illinois at Urbana-Champaign [‡]Stony Brook University

Abstract

Quorum systems (e.g., replicated state machines) are critical distributed systems. Building correct, high-performance quorum systems is known to be hard. A major reason is that the protocols in quorum systems lead to non-deterministic state changes and complex branching conditions based on different events (e.g., timeouts). Traditionally, these systems are built with an asynchronous coding style with event-driven callbacks, but often lead to “callback hell” that makes code hard to follow and maintain. Converting to synchronous coding styles (e.g., using coroutines) is challenging because of the complex branching conditions. In this paper, we present Dependably Fast (DepFast), an effective, expressive framework for developing quorum systems. DepFast provides a unique `QuorumEvent` abstraction to enable building quorum systems in a synchronous style. It also supports composition of multiple events, e.g., timeouts, different quorums. To evaluate DepFast, we use it to implement two quorum systems, Raft and Copilot. We show that complex quorum systems implemented by DepFast are easy to write and have high performance. Specifically, it takes 25%–35% fewer lines of code to implement Raft and Copilot using DepFast, and the DepFast-based implementations have comparable performance with the state-of-the-art systems.

1 Introduction

Quorum systems are critical distributed systems. In a quorum system, a node sends a request to a group of nodes, and proceeds on receiving a quorum of acknowledgements. The quorum size depends on the system design; it usually varies between a majority, a super-majority, or the whole group. The most common quorum systems are replicated state machines—a linearizable and fault-tolerant group of distributed nodes coordinating through a consensus protocol such as Paxos [39] and Raft [48]. Such quorum systems are widely deployed in practice, especially as critical infrastructures of large-scale cloud and Internet services [17, 20, 23, 34, 60].

Building a quorum system is hard. Quorum systems often have a complex consensus protocol. A node in these protocols have a complex state space. An event, e.g., a reply to a message, or a timeout, will trigger a state transition of the node. At each state, the node has multiple possible branches to go into based on the event and its current state. Though the state transition itself could be deterministic, the entire node behav-

ior is not. This is mainly because of: 1) non-determinism in some event types, e.g., timeout, 2) the inter- and intra-node parallelism, and 3) network asynchrony, e.g., message delays and out-of-order delivery.

The traditional way to code these complex state transition conditions is through an event-driven or asynchronous coding style. For each event, the developer defines an event handler, or a callback function, to drive the program to the next state. On the one hand, writing code in this style could have its benefits, mainly two-folded: 1) it could match the style of a more formal description, e.g., a TLA+ [38] specification; 2) it could have a high performance as even-driven programming is often considered to be fast in concurrent programs.

On the other hand, coding in an asynchronous style has drawbacks, mainly making the code harder to write and follow, more error-prone, and harder to debug. Since the main workflow of handling a request from begin to end is expressed in many (callback) functions as opposed to a single function, the developer needs to manually maintain the shared control, data, and debug variables across these functions. The developer also needs to manually map each function execution properly to the request’s lifetime, for example, dropping a reply if the system has moved with a quorum without the reply. Overall, this asynchronous code style could increase difficulty to develop and maintain the system. This problem is also known as stack ripping [15] or callback hell [28]).

A natural way of resolve the above issue is turning the asynchronous code into synchronous style, using lightweight solutions like coroutines. This is a well studied idea in systems research [15, 37, 52] and has been adopted widely in practice [13, 14]. However, the past study of using coroutines to inline callbacks mainly considers supporting a single asynchronous call [15, 52], e.g., inlining a single RPC callback. This is insufficient for a quorum system, which often has many concurrent callbacks and timeouts, and branches based on these concurrent event composition. Take the following behavior of a classic consensus system for example, a node sends requests to a group, then it needs to proceed to different branches based on the replies when it receives 1) a majority of acknowledgements, 2) a majority of rejects, 3) a majority of replies, mixed acknowledgements and rejects, and 4) fewer than a majority of replies. Each of these conditions have their own timeouts, and the program can only enter one branch. It is prohibitively difficult to express these conditions with a single coroutine. To make matters worse, the behavior could be even more complex with advanced protocols which have

multiple quorum sizes. For example, in fast-path enabled protocols such as Copilot [47], a super-majority quorum and a majority quorum apply at the same step in the protocol, which further complicates the situation.

In this paper, we present DepFast (Dependably Fast), a coroutine-based distributed programming framework to address the aforementioned challenges. Like prior works [15, 52], DepFast promotes synchronous code style with cooperative task scheduling and provides an Event abstraction to wrap the waiting points. The unique part of DepFast is that it provides a QuorumEvent abstraction that enables the construction of straightforward protocol descriptions, even for quorum systems with complicated timeout rules and multiple quorums (which was previously only expressible in bug-prone callback style). A QuorumEvent represents the system state of a quorum and any event that affects the state (e.g., arrival of a reply, a timeout) is funneled through the QuorumEvent. The program thus can synchronously express the conditions and branches using the QuorumEvent in a plain if-else style. Furthermore, the events are composable in DepFast, making it easier to deal with cases such as waiting for multiple quorums of different sizes at the same step.

Using DepFast to build quorum systems, the main control flow of the protocol can be written in a single function with an unripped stack. The code written in this style is easy to follow and to debug. As a case study, and as a motivation, we studied the fail-slow behavior of quorum systems, where a node in the system can be much slower than a non-faulty node but still functioning [55]. Debugging fail-slow behavior requires carefully identifying the line of code to add log statements for tracing the lifetime of a request. Writing code in the DepFast style simplifies debugging.

To evaluate DepFast, we use DepFast to implement two quorum systems based on Raft [48] and Copilot [47] respectively. We report our programming experience with DepFast. We demonstrate examples illustrating that DepFast leads to better implementations compared to the common practice of asynchronous, callback-style code; in particular, DepFast enables direct translation of the protocol algorithms and allows precise expressions of complex wait conditions, resulting in 25–35% fewer lines of code. We evaluate the performance of DepFast-based Raft and Copilot implementations against state-of-the-art versions to show our approach imposes no performance penalty. Moreover, the DepFast-based implementations have better tolerance against various types of fail-slow faults by construction, than the state-of-the-art versions.

The paper makes the following contributions:

- We design QuorumEvent to enable the construction of protocol descriptions even for quorum systems with complicated timeout rules and multiple quorum sizes.
- We develop DepFast, an effective, expressive framework that enables developers to implement complex quorum systems with a synchronous programming style.

- We show how DepFast benefits the implementation of quorum systems by illustrating DepFast-based Raft and Copilot implementations compared to state-of-the-art versions.
- We evaluate the performance of DepFast-based Raft and Copilot and show that the DepFast design for the ease of programming does not come with a performance penalty.

2 Background and Motivation

2.1 Quorum systems and async. programming

This paper uses quorum systems to refer to distributed systems with a communication pattern that requires replies from a quorum in a group of nodes. One typical type of quorum systems is a replicated state machine which uses consensus protocols (e.g., Paxos [39] and Raft [48]) to let nodes agree on the next state transition.

Consensus protocols are known to be hard to implement. Take the following behavior in Paxos and Raft for example, a node broadcasts a replication request (Accept in Paxos, or AppendEntries in Raft) to a group; then the node needs to react to events including replies and timeout. A reply can be either an acknowledgement or a reject. Based on different events, this node needs to enter different branches when it receives: 1) a majority of acknowledgements before the timeout, 2) a majority of rejects before the timeout, 3) a majority of replies, mixed acknowledgements and rejects at the timeout, and 4) fewer than a majority of replies at the timeout. After the node chooses a branch, any future event needs to be dropped. The inter- and intra-node concurrency and the non-deterministic event triggering lead to a very complex program state space.

Many formal and informal protocol descriptions of quorum systems are written in an *asynchronous* style: “upon receiving a (reply to a) message, the system acts as follows.” It is intuitive to construct the code in the same way (e.g., writing a message handler in the message loop, a callback in event-driven model, or an actor in the actor model to process a particular message). Coding in the asynchronous style has the benefit of matching a formal description of the protocol (e.g., a TLA+ [38] specification). However, it could cause the control flow to be shredded into many sub-functions like callbacks. This leads to spaghetti code, also known as stack ripping [15] or callback hell [28]. Take a Paxos system for example. For each request that goes through the 3 phases (Prepare/Accept/Commit), the main control flow will at least be shredded into 3 types of callbacks. If this is a 5-replica system, the callbacks will be executed 3×5 times. If we further count callbacks caused by disk logging (asynchronous I/O), there will be even more (at least doubled) callbacks.

It does not take long before a developer loses track of how callbacks affect each other. It also imposes significant challenges to manage the waiting process, especially in cases where the callbacks are written by different developers (which is often the case in practice). A natural way to address this

problem is to turn the code to a synchronous style by inlining the callback into the calling function. This could greatly improve understandability and maintainability. In fact, some papers describe the main control flow in this style. For example, in the Paxos paper [39], after a proposer sends out the proposal, “*if the proposer receives a response from a majority of acceptors...*” In practice, many projects started with the asynchronous callback style, but the growing size of codebase and the accompanying cognitive load of callbacks made the developers change to a synchronous code style [15, 18].

In the past, asynchronous versus synchronous programming styles (or event versus thread) have drawn many discussions [15, 19, 25, 26, 37, 44, 49]. The asynchronous or event-driven style is often preferred for performance reasons. For example, it can avoid the overhead of many OS threads, and the `epoll/kqueue` model of processing network interrupts can efficiently utilize the interrupt-based OSes and devices. Past works have proposed solutions to turn asynchronous code into synchronous and preserve its high performance, e.g., by using lightweight threads like stackful coroutines [15, 52]. However, existing solutions mainly consider inlining a single callback like a single RPC. It does not address the challenges of implementing quorum systems, where a broadcast request triggers a group of callbacks, and the system needs to proceed with a quorum.

2.2 Experience in debugging fail-slow behavior

We recently studied fail-slow behavior of real-world quorum systems [55]. Our journey started from observing that quorum systems of product-grade databases cannot meet fault-tolerance properties of the consensus protocols—a fail-slow follower affects system-wide performance. Such behavior contradicts the theory—a quorum system should proceed when there is a majority of non-faulty nodes. Specifically, a fail-slow follower should not have visible impact by design.

To reveal the root causes of the fail-slow behavior, we spent two person-years to analyze the three implementations. In our experience, debugging fail-slow fault tolerance is challenging and time-consuming. At a high level, the debugging process is a binary search for small fragments of code that caused the slowness using time stamping. The process sounds easy (as we imagined it to be), but is painful in practice. The asynchronous code often looks like spaghetti: the main control of a request is spread in different code fragments. Understanding where those code fragments are located and how they interact is non-trivial. In fact, working with developers of two of the databases, we find that even they have the same experience.

We also find that asynchronous code often lacks a clear abstraction between the quorum logic (e.g., the Raft protocol) and common, low-level utilities (e.g., RPC, disk I/O) in the spoken implementations. In addition to the challenge of expressing complex state transitions, lacking a clear abstraction has two more problems in implementation.

First, when a buggy fail-slow behavior occurs, it is hard to know whether the bug is caused by the protocol code or the utility code. A bug in the utility code is typically easier to identify and fix than a logic bug. It would be very helpful if the code can be constructed in a way that the logic code and utility code are isolated and separately profiled for debugging.

Second, a lack of abstractions also indicates lacking knowledge across the two parts. The utility code has to blindly execute the requests passed by the logic code and cannot perform optimizations to tolerate fail-slow behavior, but push the burden back to protocol logic. For example, the Raft logic broadcasts `AppendEntries` to all replicas and waits for a quorum of replies to proceed. In many existing implementations, the Raft logic sends the same message to each replica and the utility faithfully puts the message to the buffer of each replica. If one replica is slow, the connection would be slow and the buffer would keep increasing, leading to the backlog issue reported by recent work [27, 29]. If the utility is aware that this is a broadcast that can succeed with a quorum of replies, it can safely discard the messages for the slow connection.

2.3 Goal

Our experience in building and investigating complex quorum systems has driven us to rethink the programming practice and seek a more foundational solution that can make it easier to build and maintain quorum systems. We propose a synchronous programming framework for this purpose. In particular, we will propose abstractions that allow developers to implement complex quorum state transitions that can only be implemented with an asynchronous programming style before. We will show by our experiences that this framework can help programmers efficiently express complex quorum conditions in a way that is easy to follow and maintain.

3 The DepFast Framework

This section introduces the Dependably Fast (DepFast) framework. DepFast aims to provide an effective, expressive programming framework for building quorum systems. We first go through the interface of DepFast (§3.1), including an important abstraction we propose for quorum systems (§3.1.2). Then we go into internals of the framework, describing how it is implemented (§3.2).

3.1 DepFast from a programmer’s perspective

3.1.1 Coroutines and events

DepFast provides programmers with two main interfaces:

- a coroutine interface for launching tasks,
- an event interface which wraps the waiting points, or any potential fail-slow points, in the code.

The idea of using coroutine is to keep the code in one piece—avoiding callbacks, while maintaining the performance when dealing with an operation that needs to wait. For example, the code snippet below is triggering an RPC with a callback. Traditionally, it is considered the high-performance way of writing concurrent programs as it avoids the cost of creating and switching between threads. On the other hand, it comes with the cost of breaking the control flow, leading to many side effects besides increasing code complexity. For one, the programmer needs to manually maintain a shared stack from callbacks to callbacks, which is also known as *stack ripping* [15]. For a quorum-based system, the case could be more complex. For example, a reply is not always “valid”; an outdated reply needs to be ignored. The programmer needs to manually manage those, as exemplified in the code below:

```
void DoAppendEntries() {
    for (auto rpc_proxy : servers) {
        auto entries = ...;
        // the next line bears possible slowness
        auto rpc_event = rpc_proxy.AppendEntries(entries,
            AppendEntriesCallback);
    }
}
void AppendEntriesCallback(Id id, Result result) {
    // manually manage shared data
    auto reply_ok_cnt = reply_map_g[id];
    // manually manage lifetime
    auto status_ = status_map_g[id];
    if (status_ == undecided) {
        ... // only process when the log is still alive
    } // else ignore
}
```

Using coroutine, the above code can be expressed as:

```
Coroutine::Create([] () {
    vector<RpcEvent> events;
    for (auto rpc_proxy : servers) {
        auto entries = ...;
        // the rpc call is asynchronous and non-blocking
        auto rpc_event = rpc_proxy.AppendEntries(entries);
    }
    for (auto& rpc_event: events) {
        // block coroutine until the rpc returns
        rpc_event.Wait(); // possible slowness
        Process(rpc_event.Result());
    }
})
```

In the above code, what was split in callbacks is glued together in a single function, where the programmer can continue using the stack to share system states with a unified control flow. The code using coroutine can provide similar performance to the code with callbacks, because coroutines do not incur the heavy overhead of OS threads [15].

Note that the above code uses `RpcEvent::Wait()` on this object would suspend the current coroutine until the RPC has the return value ready. DepFast uses such event abstraction to wrap all the waiting conditions. With the events, a programmer can suspend/resume the coroutines. DepFast implements many built-in event types to support various operations (RPC, file I/O, etc.). With all waiting points moderated by the framework, DepFast naturally empowers more analysis (§A.2). DepFast also provides novel event types to better deal with distributed, quorum-based systems. We next introduce an important event in DepFast, `QuorumEvent`.

3.1.2 QuorumEvent

The coroutine-style code above is less efficient than the callback-style code. The coroutine-style code waits for the reply from each server sequentially with a fixed order, while in the callback-style code the callback for each server’s response can be triggered out of order. This gives the callback-style code a major benefit in performance, because it only needs to wait for the first quorum of messages to arrive. In particular, it is not affected by a fail-slow remote server. On the contrary, the coroutine-style code will be slower in performance and be affected by a fail-slow server.

To address this issue, DepFast has a special event type, termed `QuorumEvent`. The key idea is to prevent any individual fail-slow event from straggling a coroutine by combining many events together into a compound event. As the name suggests, `QuorumEvent` does not need all responses from each individual event. The usage of a `QuorumEvent` is demonstrated by the following example.

```
Coroutine::Create([] () {
    auto quorum_event = QuorumEvent();
    for (auto rpc_proxy : servers) {
        auto entries = ...;
        auto rpc_event = rpc_proxy.AppendEntries(entries);
        quorum_event.add(rpc_event);
        // no longer wait for any single event
    }
    quorum_event.Wait(MAJORITY); // wait for a majority
    ...
    // after 10s, release resources to avoid backlog
    quorum_event.Release(10000 /*ms*/);
})
```

Using the above `QuorumEvent` has two benefits. First, the replies can be received out of order and the program can proceed as soon as receiving a majority of replies. Second, the abstraction helps programmers avoid writing any code that would be blocked by any single-point fail-slow remote server. A key idea that DepFast deploys to help avoid fail-slow fault propagation is to encourage programmers to not wait on any event individually but always wait on a quorum of events when possible.

`QuorumEvent` also provides a better abstraction for resource management to help avoid the backlog issue constantly observed with fail-slow faults [27, 29]. In the code snippet above, calling `Release()` will tell DepFast to release all resources related to this event after a timeout. This is particular useful for a quorum system. It gives a deferment period for the slower response to arrive, after which the system will forcibly free all the resources related, e.g., the buffer in the RPC. Allowing a deferment period is very useful in implementing many protocols, because they often need to distinguish between the two cases: 1) the response is only a bit slow, or not at all slow but just ordered after other responses, but the host can still react to those responses which benefits the system liveness 2) the response is too slow, and the receiver should react differently (e.g., a failure recovery).

To see why using `QuorumEvent` is a better approach for resource management, consider the following alternative:

```

// Create a separate coroutine for each RPC
for (auto rpc_proxy : servers) {
    Coroutine::Create([]() {
        auto entries = ...;
        auto rpc_event = rpc_proxy.AppendEntries(entries);
        rpc_event.Wait();
        Process(rpc_event.Result());
    })
}

```

This alternative approach is both synchronous in programming style and avoids sequentially waiting on each RPC call. Actually, such a-coroutine-per-task approach is very popular in modern practices, especially in writing services with Go-lang [22]. However, in our experience this may cause problems under fail-slow behavior. If the target of the RPC is slow and cannot respond in time, the coroutine will hang in the system, waiting for the response. With new requests coming in, the hanging coroutines will accumulate, eventually exhausting system resources, as each coroutine consumes at least a memory space for its stack.

3.1.3 Other event types

In general, DepFast provides two types of events: basic events and compound events. Basic events are waiting on a task to finish, such as an RPC, a disk access, or a flag to be set, etc. A compound event is a combination of basic or other compound events. Table 1 summarizes common event types in DepFast and their trigger conditions.

Basic events. One common basic event is `ValueEvent`. This is a holder for a value to be set. If the value is set to match the target value, the event will be triggered. We find this event abstraction very useful because we often find statements in algorithms written like “wait for X to become Y.” For example, in Copilot [47], to decide the execution order of a command, the system needs to wait until the status of a selected group of commands to become “committed”. Traditionally, this could be hard to implement as it would involve complex callbacks or thread synchronization. With DepFast, statements of this type can be directly translated to one line of code.

Another basic event type is `IOEvent`. We use it to wrap all synchronous I/O operations, mainly disk-related operations. An `IOEvent` corresponds to a task executed in an I/O thread. For example, the program initiates a disk write through DepFast’s interface, the actual disk operations involving `fwrite` and `fsync` will be executed in the I/O thread. The program then waits on a `DiskEvent` returned by DepFast. When the disk operation finishes, the I/O thread will notify the scheduler that the event is ready. The synchronization between I/O threads and scheduler is the only part in a DepFast program that has multi-thread synchronization. We believe that this small footprint of multi-thread synchronization can minimize the possible fail-slow issues caused by multi-threading.

Compound events. For compound events, an example is `QuorumEvent` (§3.1.2). It takes many events (e.g., `RPcEvent`) as its subevents, and wait for at least a defined quorum of

Event	Trigger Condition
<code>ValueEvent</code>	If the value is set and matches the target (it supports customized comparators).
<code>DiskEvent</code>	If the disk access operation (e.g., <code>fread</code> , <code>fwrite</code> , and <code>fsync</code>) is finished.
<code>RPcEvent</code>	If the RPC call has returned.
<code>QuorumEvent</code>	If a quorum has reached (typically used together with <code>RPcEvent</code>).
<code>AndEvent</code>	If all subevents have been triggered.
<code>OrEvent</code>	If any subevent is triggered.

Table 1: The built-in events in DepFast

them to be triggered. Other compound events in DepFast includes `AndEvent` and `OrEvent`. As the name suggests, an `AndEvent` is triggered when all of its subevents are triggered; an `OrEvent` is triggered as soon as one of its subevents is triggered. Note that events can be nested: for instance, an `AndEvent` can contain many `QuorumEvents` as its subevents.

Nesting events can express complex waiting conditions. For example, one can use an `OrEvent` to combine these 3 events: 1) a `QuorumEvent` that waits for a majority of okays. 2) A `QuorumEvent` that waits for a minority-plus-one rejects. 3) A `TimeoutEvent`. This compound event can be used to effectively catch conditions in classic consensus protocols. In fact, as we find that the abstraction is very commonly used, we have merged these three conditions into `QuorumEvent` so it has three outcomes: `Ready()`, `Fail()`, `Timeout()`,

3.1.4 A showcase of DepFast’s expressiveness

DepFast can effectively express many complex behaviors of quorum systems, organize the main control flow in a clean way, and process the complex state transitions automatically in the background. We demonstrate the expressiveness of DepFast using the code snippet of our Copilot implementation built with DepFast in Figure 1. Copilot is one of the protocols that leverages a “fast-path quorum” [40, 45, 57]. In these protocols, after broadcasting a round of (`FastAccept`) messages, there are at least three concurrent conditions to decide how the system proceeds: 1) to a fast path if receiving a super-majority of acknowledgments with identical speculative information, 2) to a slow path if receiving a majority of acknowledgments, and 3) to failure recovery if neither the above is possible (e.g., when receiving a majority of rejects, or not enough messages after a timeout). Figures 1(a) and (b) show the code and control flow chart of Copilot built on top of DepFast. As demonstrated, the code implements the protocol in a clean manner. What DepFast handles in the background is shown in Figure 1(c). Upon every event (message arrival and timeout), DepFast processes the subtle state transitions and drives the main control flow forward to the proper next state. Without DepFast, one needs to carefully implement all state transitions and error handling manually, a complex and error-prone process. With DepFast, one can build the system cleanly, with code easy to follow.

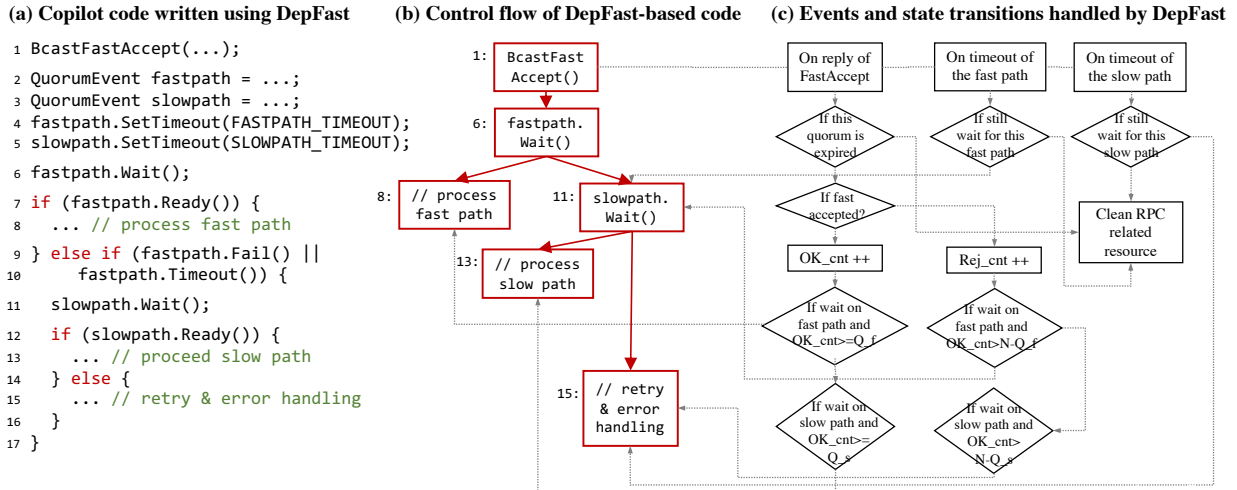


Figure 1: Expressiveness of DepFast demonstrated by DepFast-based Copilot implementation: (a) code, (b) control flow, and (c) events and state transitions. The code logic is explained in Figure 3.

3.2 DepFast internals

3.2.1 Architecture

Figure 2 shows the architecture of DepFast. A DepFast process has two major types of threads: worker threads and system threads. The former run user code and the latter run background activities.

All user-defined tasks run in worker threads as stackful coroutines; we use Boost::coroutine2 [13] as a building block. A worker thread runs an `epoll` [6] (or `kqueue` [10] for BSD-based systems) loop that wakes up on network interrupts or timeouts. The system has a built-in high-performance RPC module that uses the `epoll` for incoming and outgoing messages similar to other high-performance networking library like `libev` [11]. The RPC module provides automatic client code and server handler header generation from an RPC declaration file (like `gRPC` [9]). The RPC works asynchronously and provides synchronous event binding (`RpcEvent`).

Having the RPC working asynchronously allows us to avoid launching a separate coroutine for each RPC (§3.1.2). Take the `QuorumEvent` for example. The system only has one global `epoll` loop that receives replies for all RPCs. Once the system receives a reply, it is matched to the belonged `QuorumEvent`, and a counter is updated in that `QuorumEvent`. Once that counter reaches the quorum number, the `QuorumEvent` is ready, and then the system resumes the coroutine waiting on it. There is no coroutine creation during handing a message that might contribute to a quorum.

In the same worker thread runs the scheduler functions. The scheduler is in charge of managing user coroutines. All coroutine lifetime related functions, including coroutine creation, deletion, pause, and resume, are provided by the scheduler. The scheduler performs a check at each `epoll` wakeup. It checks whether the event a coroutine is waiting on is triggered

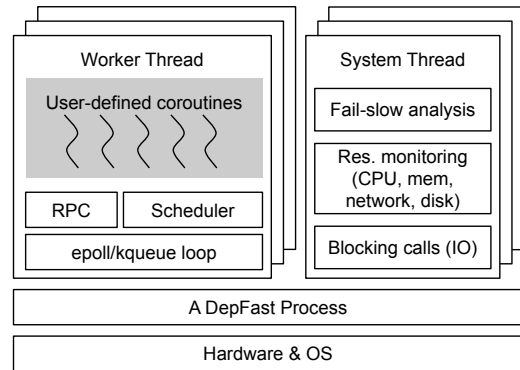


Figure 2: The architecture of DepFast

(or timed out), and then resume the paused coroutine.

While the worker threads handle most of the system functions on the critical path, the system threads deal with the additional features of the framework. Among these, a significant function is to support blocking calls. Examples include disk flush calls (`fsync`) and support for third-party libraries. DepFast supports these by wrapping them in system threads. Other features that run in system threads include the resources usage monitoring (CPU, memory, network, disk) and fail-slow analysis, which we will discuss later.

3.2.2 Lazy and cooperative scheduling

The base class of an `Event` is implemented as follows:

```
class Event {
    // timeout defines the maximum duration of waiting
    // on the event. -1 means waiting forever.
    void Wait(int timeout = -1) {
        if (IsReady()) return;
        if (timeout >= 0) {
            sched_s.sorted_timeouts.insert(now()+timeout);
        }
        // sched_s is a thread_local static variable
        sched_s.yield_current_coroutine(this);
    }
}
```



```

void Test() {
    if (IsReady()) {
        // the scheduler will put the suspended coroutine
        // back to the ready_coroutine queue.
        sched_s.notify_ready(this);
    }
}
// To implement by specific event type to define when
// this event is ready to resume suspended coroutine.
virtual bool IsReady();
}

```

The coroutine management operates in a lazy and cooperative approach. It does not preemptively suspend or resume a coroutine. When a coroutine is running, it fully occupies the worker thread. A key structure in the scheduler is a queue that records pending events and the corresponding suspended coroutines. When a coroutine runs, it may change the status of other events (e.g., changing the value of `ValueEvent`). When this happens, the scheduler will *not* yield the current running coroutine and switch to the resumable coroutine immediately, but mark the event ready and put it in a ready queue. Only when the running coroutine calls `Wait` on an event, the scheduler is triggered to do scheduling work.

The scheduler will first check if the event is ready (i.e., no wait is necessary). If so, the `Wait` call returns directly without yielding out the coroutine; the caller coroutine continues running. If not, the scheduler will put a reference of the event and the caller into a pending queue, and then looks for the next coroutine to run. The scheduler first checks the queue of ready events and resumes them one at a time. Note that the queue of ready events may grow during this process, as the resumed coroutine may turn more events ready. After the ready queue is empty, the scheduler searches the pending queue for events that are timed out, and resumes the suspended coroutines correspondingly. To make the search faster, the pending queue is sorted based on the timeout timestamps.

Reversed backlog. In our development of DepFast, we encountered an interesting issue termed “reversed backlog.” When a system has a slow node, RPCs to this slow node will be delayed. However, the delays are not necessarily uniform over time. The responses often arrive in a burst pattern—the RPC sender will not hear back from the slow node for a while, and then suddenly receive many responses from it. In our early implementation, the `epoll` loop would process everything available from a connection before moving on to the next. This caused the system to occasionally hang on processing the outdated RPC responses. To deal with this issue, the system is improved to process data from connections with a round-robin approach; it will pause processing a connection after reaching a threshold to avoid starving other connections.

3.2.3 Concurrency and multi-threading

Through the coroutine model, DepFast allows multi-task concurrency inside a single worker thread. Similar to many other asynchronous event or coroutine frameworks, DepFast encourages programmers to exploit concurrency in a single worker

thread before moving to multi-threading. The advantage of running tasks in a single thread is to avoid thread-safety issues. To DepFast, an extra benefit is to eliminate the possibility of fail-slow faults caused by thread locks, a known suspect of performance issues. In reality, running tasks concurrently with a single thread can give high enough performance in most cases, as shown in our evaluation (see §6.2 and §6.3).

Moving to multiple threads, to write thread-safe code the users need to either shard the system into different threads and regulate inter-thread communication, or write memory-sharing code and use mutexes for mutual exclusion. DepFast encourages the former as it minimizes the chances of performance issues caused by waiting on mutexes.

4 Discussion

We discuss the software engineering benefits of using DepFast to build quorum systems. We exemplify the challenges in implementing complex quorum conditions and discuss how DepFast can address the challenges. As an concrete example, the following code snippet processing a fast path is taken (and simplified) from the EPaxos implementation [3], a popular academic prototype of advanced consensus protocols.

```

func handlePreAcceptReply(reply) {
    inst := ... // find consensus instance
    ... // return if this is a delayed request
    if reply.OK {
        inst.preAcceptOKs++
    } else {
        inst.preAcceptRejects++
        if inst.preAcceptRejects >= r.N/2 {
            // TODO
        }
    }
    fastpathSatisfied = ... // test fastpath conditions
    if inst.preAcceptOKs >= N/2 && fastpathSatisfied {
        // proceed to fast path
    } else if inst.preAcceptOKs >= N/2 {
        // proceed to slow path
    }
    //TODO: take the slow path if msgs are slow to arrive
}

```

We can see two TODOs in the code. The first TODO is on counting rejects. If a node receives too many rejects to proceed, it should enter error handling to retry this request. In DepFast, this maps to the branch where `slowpath.Fail()` happens. The second TODO is when messages are slow to arrive, this case maps to DepFast’s branch where `fastpath.Timeout()` and/or `slowpath.Timeout()` happen.

Implementing the two TODOs would require heavy revisions on the code logic and change the control flow, as it cannot be done by naturally replacing the TODO comments with two function calls. In contrast, with DepFast, one can implement both TODOs in place. For the first TODO, a retry function can be synchronously called in the reject branch. For the second TODO, if messages are slow to arrive, a function calling the slow path can be put right in the timeout branch.

In fact, TODOs are not the only problem in the code. The fast-path condition is also simplified and suboptimal. The

condition is important, because it decides whether to enter a fast path or a slow path. In the code, once it sees a majority of OK replies, it makes the decision based on the current calculation of the conditions (`fastpathSatisfied`). However, this decision could be suboptimal, because the conditions could change if more replies arrive. As a consequence, a node loses the opportunity to enter a fast path if it waits a little longer, but directly enters the slow path. The simplification, despite being suboptimal, is understandable, because it is very hard to express the conditions accurately in the asynchronous code style. With DepFast, this can be expressed easily with the timeout scheme on a `QuorumEvent`.

Lastly, the conditions assume that the fast- and slow-path quorums have equal size, which indicates that it only works for at most 5 replicas. If the replication group size is bigger [4], the code needs heavy revisions. Simply replacing $N/2$ in the `if` branch with a larger super-majority value is incorrect: the code will always choose a slow path because the `else if` branch will always be taken.

Although this discussion is based on EPaxos, we find that the conditions are error-prone in other quorum system implementations, especially regarding timeout handling. For example, CockroachDB had a bug caused by having no timeout on lease acquisition [5], which may lead to system stalling. In DepFast, because timeout can be easily added to an event, such problems can be prevented. In fact, we often use this practice for debugging: for cases that cause the system to stall, we add a global default timeout to all events, and then we can easily find the stall point of the problematic code.

5 Building Quorum Systems with DepFast

To demonstrate the usefulness and effectiveness of DepFast, we use DepFast to build two quorum-based systems: Raft [48] and Copilot [47], named as DepFast-Raft and DepFast-Copilot respectively. In this section, we discuss our experiences in using DepFast to build these systems, with a focus on how to “translate” the protocol algorithms into system implementations effectively.

5.1 DepFast-Raft

Raft’s protocol largely consists of two parts: 1) *leader election*: a candidate broadcasts `RequestVote` RPCs to all the other servers; it becomes a leader once votes from a majority of servers are received. and 2) *data replication*: for each follower, the leader keeps a mark (`nextIndex`) of the next log position to send to that follower; if the follower is lagging (due to out-of-order messages, network issues, etc.), the leader repeatedly sends the log entries needed by the follower.

Leader election can be effectively expressed with DepFast’s `QuorumEvent` design: a server broadcasts requests to other servers and can proceed after it receives a quorum of acknowledgments. Data replication, though described in a different

style in the Raft paper [48]— from a follower’s view, not a quorum’s view—can also be expressed to the same pattern above. Our implementation uses one coroutine to initiate the broadcast of the `AppendEntries` requests and wait for a quorum of responses. As DepFast handles most of the complexity in the network, disk, and event processing, a Master student was able to translate Raft’s pseudocode directly into a stable C++ implementation (used in §6.2) in ten days. The implementation has $\sim 1,200$ lines of code. As a rough comparison, the Raft logic in etcd [8] is implemented in $\sim 1,600$ lines of code in Go; braft [2], an open-source Raft implementation in C++, has $\sim 3,500$ lines of code.

An interesting case we found in implementing DepFast-Raft relates to the way Raft describes its protocol. In the design of Raft’s data replication protocol (its `AppendEntries` RPC), if the follower is lagging, the leader repeatedly sends log entries that are missing on the follower. This design is optimized for lagging servers and for new servers trying to catch. However, the way the algorithm is described may naturally lead to an implementation with separate threads synchronizing with different followers. A natural implementation could split the code responsible for committing a request into different functions in different threads. This style of implementation works well when there are no failures, but could take more time to debug when there are unexpected fail-slow behaviors, because it requires more work to trace the progress of each request. Our implementation, instead, uses a single coroutine to initiate the broadcast of the `AppendEntries` requests and wait for a quorum of responses. In case of an occasional reject due to the follower being lagging, the leader will launch a background coroutine, which is off the critical path of client requests, to synchronize with the lagging follower with additional `AppendEntries`.

5.2 DepFast-Copilot

Copilot is a consensus protocol that tolerates any single fail-slow node including the leader. It has two leaders, a pilot and a copilot, each is the backup of the other in case one fails. Copilot’s complexities mainly rise from following designs:

- *Commands ordering*. Each leader maintains two separate logs, one for itself and one as the backup of the other leader. Copilot’s ordering protocol coordinates between pilot and copilot to determine the *dependencies* of log entries, which specifies the prefix of the other log that should be executed before a given log entry.
- *Commands execution*. Unlike Raft that uses an index to order command execution, Copilot’s execution order is more complex: (1) Copilot has a protocol that calculates the right order of a command based on its dependencies. The calculation process is restricted by the status of these dependencies, e.g., the execution must happen after all the dependencies satisfy a rule; (2) The execution of the command is also constrained by the dependencies, i.e., a command’s execution

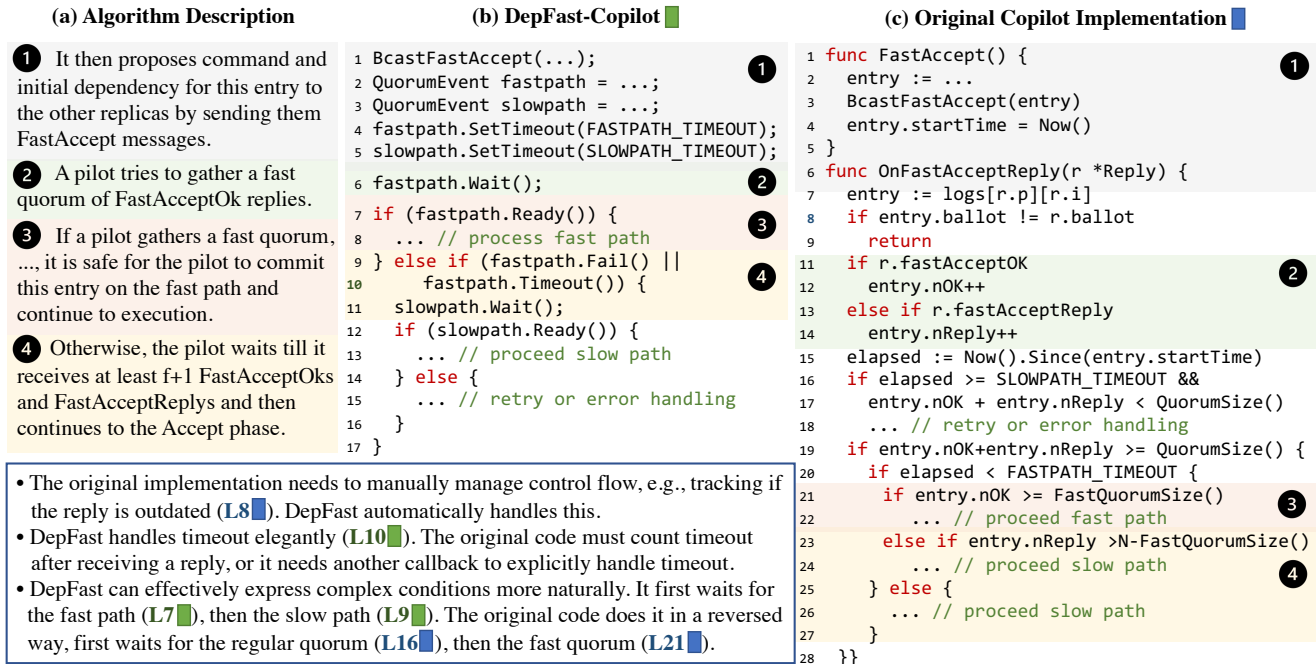


Figure 3: Comparison of the Copilot implementation using DepFast and the original implementation (both are simplified)

must wait until all the predecessors are executed.

- Fast takeover.* When one pilot becomes slow or fails, the other pilot needs to take over entries in the slow pilot’s log to prevent waiting for its commit for too long. It broadcasts Prepare to all other replicas to collect the entries and their status on other replicas at that position. Depending on the replies collected, Copilot distinguishes between many different cases to choose an entry properly, including whether there are committed entries, how many entries are fast-accepted, how many entries are accepted, etc.

As we have discussed in Section 3.1.4, DepFast can effectively express Copilot’s complex behaviors. Figure 3 shows a comparison of the DepFast version and the original version of Copilot. As shown, the DepFast version is closer to the algorithm in flow, and is easier to follow.

The commands execution algorithm contains statements like “waiting for the commit/execution of”, which is common in other protocols that use dependency for commands ordering (e.g., EPaxos [45]). However, such kind of behavior is not straightforward to express in an asynchronous programming style. In fact, the original Copilot implementation adopts this asynchronous programming style. It uses a separate Goroutine to keep scanning through the log and breaks the loop to start from the beginning when the above waiting condition is not satisfied. Instead, DepFast’s Wait API captures such behavior effectively. What we do is to represent the commit/execution of a log entry as an event and call Wait if there is a dependency on it as shown in Figure 4.

We find that the Copilot implementation using DepFast is more concise and readable than the original asynchronous,

callback-style implementation. To give a rough, unsolicited idea, the original implementation of the core protocol (excluding the utility code) has ~2,500 lines of Go code [12]. DepFast-Copilot only has ~1,600 lines of C++ code, despite that C++ is less expressive than Go.

Anecdotally, we started a Copilot implementation without DepFast, using an asynchronous callback style. In the process we ran into a bug that *sometimes* froze the system, which was caused by a wait condition being not triggered properly. We find bugs of this type are very hard to debug (we spent two weeks debugging it) because there is not a simple way to track each wait condition. We re-implemented the wait conditions using DepFast’s Wait API. The new implementation was done in roughly two days and we never encountered the same problem. Thanks to DepFast’s coroutine and event model, it is very easy to find out which event the system is waiting on, and print all the stack frames of the suspended coroutine.

6 Evaluation

We have shown the software engineering benefit of DepFast is its expressiveness and programmability (§3.1.4, §4 and §5). In evaluation, we mainly focus on answering two questions: 1) Does the expressiveness come at a cost of performance, or, can systems implemented in DepFast achieve the same level of performance as heavily optimized production and academic systems? 2) Can DepFast help system implementations guarantee their fault tolerance? This section answers the two questions by comparing Raft and Copilot implemented in DepFast to etcd and the original Copilot implementation. Specifically,

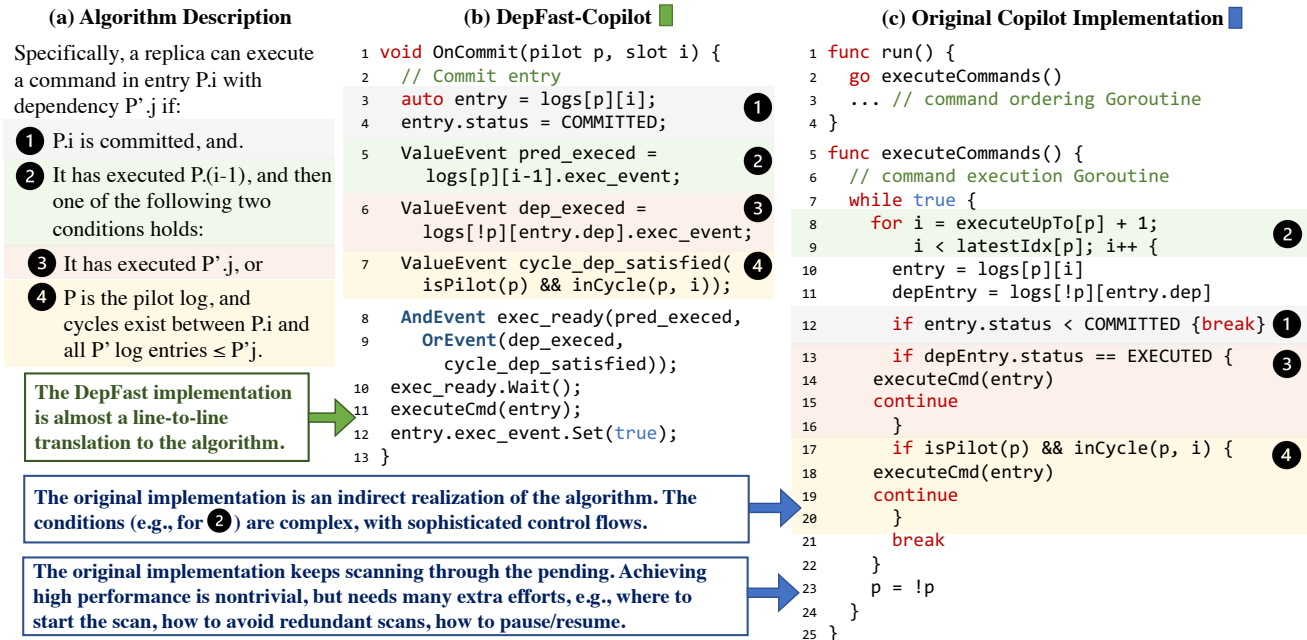


Figure 4: Comparison of the Copilot implementations using DepFast and the original impl. (both are simplified), cont'd

for fault tolerance, we evaluate DepFast-based systems on *fail-slow* fault tolerance using fail-slow faults [27, 31].

6.1 Experiment Methodology

We ran all the experiments on Azure Cloud. For each system, we evaluated it on a 3- and 5-replica cluster: each node runs on a Standard_D4s_v3 virtual machine (VM), with 4 vCPUs, 16GB RAM, and 64GB SSD. The server process is bind to one CPU core for all the systems evaluated. We ran the clients in a Standard_D16s_v3 VM, with 16 vCPUs and 64GB RAM.

Workloads and metrics. For all the evaluated systems, including our Raft and Copilot implementations, and the reference implementations (etcd [8] for Raft and the original Copilot implementation [12]), we use a single K-V, 100% write workload. We measure performance metrics, including throughput and latency distribution. Each trial runs for 120s and is repeated for 3 times. We display the results with the median throughput of the 3 trials, with the error bars showing the deviation.

Configurations. We use quorum reads and writes in our evaluation. We use the load that reaches the max CPU utilization on leader for the fault-injection experiments, marked with red stars in Figures 5(a) and 6(a). For DepFast-Raft, the server replies to the client after the log entry has been persisted on disk. For DepFast-Copilot, we set the fast-takeover timeout to 10ms and the command batching timeout to 1ms (same as the original Copilot implementation).

Fault Injection. We build a fail-slow fault injection testing tool to inject different types of fail-slow faults on system components (including CPU, memory, SSD, and network in-

terface) into the target systems and measure their impact in terms of the end-to-end performance. The fail-slow faults are simulated based on prior studies on fail-slow faults and represent common fail-slow modes [27, 31]. Table 2 describes those faults and the corresponding injection methods. For DepFast-Copilot, we do not inject faults on the disk, because the implementation is memory-based.

We inject fail-slow faults in the way that is expected to be tolerated without losing throughput by the consensus protocols of the target quorum systems. For DepFast-Raft, we inject faults to a minority of followers [55]. For DepFast-Copilot, we inject faults to a minority of nodes that can include one of the leaders [47].

6.2 DepFast-Raft

Figure 5(a) shows the latency and throughput of DepFast-Raft and etcd with both 3- and 5-replica setups. When binding the server process to one core, our DepFast-Raft achieves a maximum throughput of over 20K and 18K RPS (requests per second) with 3- and 5-replica setups, respectively; etcd has a peak throughput of 8K RPS. Claiming DepFast-Raft is better than etcd would be perhaps unfair as etcd is a production system with many features. However, this test at least proves that DepFast can be used to implement systems of production-level performance, with a class-project level of building difficulty.

Figures 5(b)–(d) show the fail-slow fault tolerance of DepFast-Raft in terms of throughput, median and P99 tail latency (in CDF) in both 3- and 5-replica setups. For the 3-replica setup, we inject fail-slow faults to one follower; for the 5-replica setup, we inject fail-slow faults to two followers (the

Fail-slow Type	Injection Method	Follower	Leader
Slow CPU	Use <code>cgroup</code> to limit DB process to utilize only p of CPU period	$p=5\%$	$p=50\%$
CPU Contention	Custom program (to consume cpu) assigned $t \times$ cpu share as the DB process	$t=15$	$t=1$
Slow Disk	Use <code>cgroup</code> to limit the disk I/O bandwidth available for DB to bw	$bw=128KB/s$	N/A
Disk Contention	Use program(<code>dd</code>) to do write operation on disk while DB is running	no parameter	N/A
Slow Network	Add a delay of d to the network interface using <code>tc</code>	$d=40ms$	$d=40ms$
Memory contention	Use <code>cgroup</code> to set the maximum amount of user memory for DB process to s	$s=50MB$	$s=250MB$

Table 2: Fail-slow faults used in the measurement study and our evaluation

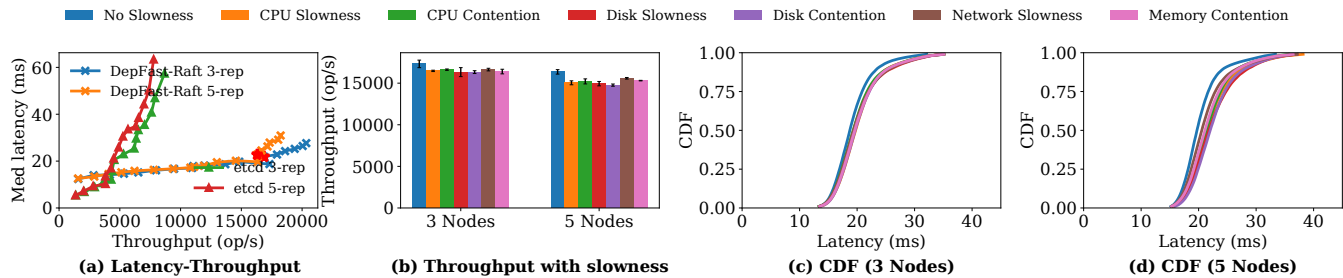


Figure 5: Performance of DepFast-Raft with various fail-slow faults on follower(s) in the 3- and 5-replica deployments.

minority). We choose to only inject faults on the followers, because slow followers in Raft should not affect performance, while the slow leaders do.

We can see that DepFast-Raft consistently tolerates the injected fail-slow faults. The throughput differences are within 6% and 10% for 3- and 5-replica setup, respectively. The differences in both median and P99 latency are within 15% range. The results are comparable to etcd under the same test (see Figure 8, §B).

We attribute DepFast-Raft’s fault tolerance to the use of the DepFast framework to manage potential fail-slow points with the event interface. Specifically, we wrap every set of RPCs with `QuorumEvent` to avoid blocking on a single slow follower, preventing slowness propagation. Besides, blocking operations (e.g., disk I/O) are wrapped and put on a separate thread to avoid blocking the main worker thread.

Note that fail-slow followers inevitably have an impact, as the system is more susceptible to network and disk I/O spikes. In a 3-replica quorum, when one follower fails slow, the quorum reads and writes can be affected by the spikes of the other follower, affecting the tail latency.

6.3 DepFast-Copilot

Figure 6(a) shows the latency and throughput of DepFast-Copilot and the original Copilot implementation [12] in a 3-replica setup. (the 5-replica results can be found in B). The peak throughput of DepFast-Copilot is 33K RPS with a 3-replica setup. The performance is comparable to the original Copilot, which is just 9.3% higher at 36K RPS. The throughput is much higher than Raft, mainly because there is an optimization in Copilot called ping-pong batching that batches many requests into one command.

We inject fail-slow faults to both a leader and a follower, as Copilot is designed to tolerate any one of the fail-slow nodes.

Fail-slow follower. Figure 6(b) and 6(c) show the throughput and latency CDF of DepFast-Copilot with a fail-slow follower. Similar to the results of DepFast-Raft (§6.2), the faults do not have a significant impact on system-wide performance. The differences in throughput are within 10%, and the differences in median latency and P99 latency are within 9% and 30%, respectively. We observe that DepFast-Copilot’s tail latency is more susceptible to fail-slow faults than that of DepFast-Raft. Apart from the spikes discussed in §6.2, we attribute that to Copilot’s property of having two leaders. With a maximum number of minority followers being slow, a reply from another leader must be obtained to form a quorum, while with fewer slow followers a leader can form a quorum just with replies from non-slow followers. However, the load on a leader is much higher than a follower, causing the tail latency of replies from a leader node to be higher than replies from a follower node. That in turn results in longer tail latency to form a quorum, rendering the tail latency higher.

Fail-slow leader. Figure 6(b) and 6(d) shows the throughput and latency CDF of DepFast-Copilot with a fail-slow leader in a 3-replica setup. With Copilot’s multiple-leader design, there is no significant impact on throughput (the differences are within 25%) and the increase in median and P99 latency are within a reasonable range under the existence of one slow leader. For DepFast-Copilot, the CPU contention has the most significant impact on tail latency, in which case the P99 latency increased by 10ms.

The original Copilot implementation. We did the same fault-injection experiments on the original Copilot implementation (results are in Figure 9, §B). We can successfully duplicate the results in its paper that Copilot can tolerate a node slowdown in a slow network simulation. In some other cases (CPU, memory), we find that the original Copilot implementation cannot tolerate the failures as well as DepFast-Copilot. On one

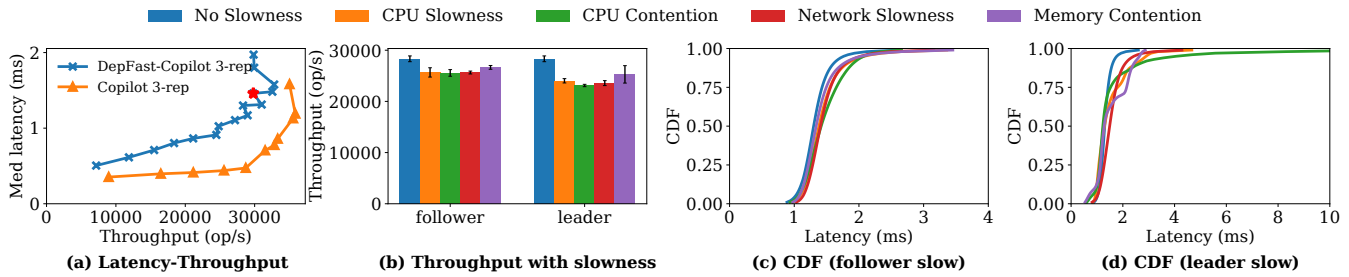


Figure 6: Performance of DepFast-Copilot with various fail-slow faults on follower(s) and leader in the 3-replica deployment.

hand, this is justifiable as the original Copilot implementation is an academic prototype that focuses on verifying the design, rather than the implementation problems. On the other hand, it proves that fault tolerance is not only a design problem but also an implementation problem. It further proves the value of having another layer like DepFast in the system.

7 Related Work

Synchronous versus asynchronous programming. The discussion on synchronous and asynchronous programming styles started decades ago [41]. As common wisdom, synchronous programming (using threads instead of callbacks) is easy to follow, but tends to have unstable performance due to the overhead of OS threads [15, 25, 26, 37, 42, 43, 52, 53]. Prior work on synchronous programming focuses on reducing its overhead using cooperative task scheduling with lightweight user-space threads (e.g., coroutines or fibres) [15, 43, 52, 53]. Today, coroutines and cooperative task scheduling have been widely accepted, with built-in support in modern languages such as Go and C++. Our work follows the same principles, and extends the literature by considering distributed systems code (prior work focuses on I/O operations on a single node). An orthogonal direction is to improve the understandability of callback-style code, making it synchronous code alike [25, 37]. The approach needs compiler support and extra tooling.

Distributed programming patterns and frameworks. The actor model is a common model to construct distributed programs, e.g., Erlang/OTP [7] and Scala/Akka [1], Orleans [21] and ActOp [46]. Our target is not actor systems, as we focus on the imperative coding style with RPC for communication that is still a common practice in building C/C++ system software. But our results can be complimentary to the actor world, because the actor systems are mostly asynchronous, and may be subject to the same callback hell problem [56].

Rex [32], Eve [35], and Crane [24] target fault tolerance at the OS process level. Ambrosia [30] extends the actor model with built-in fault tolerance support. These frameworks are potential users of DepFast—they can use DepFast to build fault-tolerance mechanisms and services.

A few frameworks help programmers to match their implementations to the specifications rigorously and thoroughly.

Mace [36] translates specification into a C++ implementation and provides a model checker to verify correctness. Rules-based programming [50] promotes programming in an event-based state machine, which helps specify concurrent and non-deterministic conditions. Verdi [54] and Ironfleet [33] help build formally verified systems. DepFast has a different goal: making distributed system code easier to write, maintain, and debug. It also addresses different issues, e.g., fail-slow fault propagation and backlogs (reported in formally verified quorum implementations [29]). DepFast also imposes much fewer restrictions on how distributed systems are programmed.

8 Concluding Remarks

We have presented DepFast, a programming framework to build quorum systems. Our experience of using DepFast is encouraging. DepFast helped us to effectively develop high-performance, fault-tolerant quorum systems with complex consensus protocols. With DepFast, we can write quorum systems code that is easy to follow and maintain. Our future work includes using DepFast to build different types of distributed systems, such as sharded datastores with distributed transaction protocols which also have complicated waiting conditions. We will investigate adopting DepFast’s abstraction with other frameworks and interfaces, e.g., C++ 20’s coroutine interface, and the actor model in Erlang/Scala.

Acknowledgments

We would like to express our deep appreciation to our shepherd, Jon Howell, who was very responsive during our interactions with him and provided us with invaluable suggestions, which have fundamentally improved this paper and strengthened our work. We also thank the anonymous reviewers for their feedback. We thank the authors of Copilot, especially Khiem Ngo, for the discussions and reviews. We thank Dan Plyukhin for discussions that helped us understand the actor programming model deeper. We thank our industry collaborations for the discussions, especially Ye Ji (CockroachDB) and Siyuan Zhou (MongoDB). This work was supported in part by NSF CNS-2130590 and CNS-2130560, and Microsoft Azure credits.

References

- [1] Akka. <https://www.akka.io/>.
- [2] braft. <https://github.com/baidu/braft>.
- [3] CMU-efficient/EPaxos, func handlePreAcceptReply. <https://github.com/efficient/epaxos/blob/791b115669fca472d3136f6a2eda46c00b3f8251/src/epaxos/epaxos.go#L1000>.
- [4] CMU-efficient/EPaxos, issue 10. <https://github.com/efficient/epaxos/issues/10>.
- [5] CockroachDB/cockroach pull request 81136. <https://github.com/cockroachdb/cockroach/pull/81136>.
- [6] epoll(7) — linux manual page. <https://man7.org/linux/man-pages/man7/epoll.7.html>.
- [7] Erlang/OTP. <https://www.erlang.com/>.
- [8] etcd. <https://etcd.io/>.
- [9] gRPC. <https://grpc.io/>.
- [10] kqueue(2) - openbsd manual pages. <https://man.openbsd.org/kqueue.2>.
- [11] libev. <http://software.schmorp.de/pkg/libev.html>.
- [12] Princeton-sns/Copilot. <https://github.com/princeton-sns/copilot/blob/main/src/copilot/copilot.go>.
- [13] The Boost Library: Coroutine2. <https://github.com/boostorg/coroutine2>.
- [14] The Go Programming Language. <https://go.dev>.
- [15] ADYA, A., HOWELL, J., THEIMER, M., BOLOSKY, W. J., AND DOUCEUR, J. R. Cooperative Task Management without Manual Stack Management. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'02)* (June 2002).
- [16] ARZANI, B., CIRACI, S., CHAMON, L., ZHU, Y., LIU, H. H., PADHYE, J., LOO, B. T., AND OUTHRED, G. 007: Democratically Finding the Cause of Packet Drops. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)* (April 2018).
- [17] BALAKRISHNAN, M., FLINN, J., SHEN, C., DHARAMSHI, M., JAFRI, A., SHI, X., GHOSH, S., HASSAN, H., SAGAR, A., SHI, R., ET AL. Virtual Consensus in Delos. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (October 2020).
- [18] BOLOSKY, W. J., DOUCEUR, J. R., ELY, D., AND THEIMER, M. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *Proceedings of the 2000 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'00)* (June 2000).
- [19] BOUCHER, S., KALIA, A., ANDERSEN, D. G., AND KAMINSKY, M. Lightweight Preemptible Functions. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'20)* (July 2020).
- [20] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Conference on Operating Systems Design and Implementation (OSDI'06)* (Seattle, WA, USA, November 2006).
- [21] BYKOV, S., GELLER, A., KLIOT, G., LARUS, J. R., PANDYA, R., AND THELIN, J. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)* (October 2011).
- [22] CHANG, S. S. Leveraging Go Concurrency. *Go Web Programming, Chapter 9, Manning Publications Co.* (July 2016).
- [23] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's Globally-Distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)* (Hollywood, CA, USA, 2012).
- [24] CUI, H., GU, R., LIU, C., CHEN, T., AND YANG, J. Paxos Made Transparent. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'15)* (October 2015).
- [25] CUNNINGHAM, R., AND KOHLER, E. Making Events Less Slippery with eel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS'05)* (June 2005).
- [26] DABEK, F., ZELDOVICH, N., KAASHOEK, F., MAZIERES, D., AND MORRIS, R. Event-driven Programming for Robust Software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop (EW 10)* (July 2002).
- [27] DO, T., HAO, M., LEESATAPORNWONGSA, T., PATANANAKA, T., AND GUNAWI, H. S. Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SOCC'13)* (October 2013).
- [28] EDWARDS, J. Coherent Reaction. Tech. Rep. MIT-CSAIL-TR-2009-024, Computer Science and Artificial Intelligence Laboratory, June 2009.
- [29] FONSECA, P., ZHANG, K., WANG, X., AND KRISHNAMURTHY, A. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the 11th ACM European Conference on Computer Systems (EuroSys'17)* (April 2017).
- [30] GOLDSTEIN, J., ABDELHAMID, A., BARNETT, M., BURCKHARDT, S., CHANDRAMOULI, B., GEHRING, D., LEBECK, N., MEIKLEJOHN, C., FAROOQ, U., NEWTON, R., GHOSH, R., ZACCAI, T., ZHANG, I., GOLDSTEIN, J., ABDELHAMID, A., BARNETT, M., CHANDRAMOULI, B., GEHRING, D., LEBECK, N., MEIKLEJOHN, C., MINHAS, U. F., NEWTON, R., PESHAWARIA, R. G., ZACCAI, T., AND ZHANG, I. A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications. *Proceedings of the VLDB Endowment 13*, 5 (January 2020).
- [31] GUNAWI, H. S., SUMINTO, R. O., SEARS, R., GOLLIHER, C., SUNDARARAMAN, S., LIN, X., EMAMI, T., SHENG, W.,

- BIDOKHTI, N., MCCAFFREY, C., SRINIVASAN, D., PANDA, B., BAPTIST, A., GRIDER, G., FIELDS, P. M., HARMS, K., ROSS, R. B., JACOBSON, A., RICCI, R., WEBB, K., ALVARO, P., RUNESHA, H. B., HAO, M., AND LI, H. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)* (February 2018).
- [32] GUO, Z., HONG, C., YANG, M., ZHOU, D., ZHOU, L., AND ZHUANG, L. Rex: Replication at the Speed of Multi-Core. In *Proceedings of the 9th ACM European Conference in Computer Systems (EuroSys'14)* (April 2014).
- [33] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)* (October 2015).
- [34] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'10)* (Boston, MA, June 2010).
- [35] KAPRITSOS, M., WANG, Y., QUEMA, V., CLEMENT, A., ALVISI, L., AND DAHLIN, M. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)* (October 2012).
- [36] KILLIAN, C. E., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. M. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)* (June 2007).
- [37] KROHN, M., KOHLER, E., AND KAASHOEK, M. F. Events Can Make Sense. In *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX ATC'07)* (June 2007).
- [38] LAMPORT, L. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (May 1994).
- [39] LAMPORT, L. Paxos Made Simple. *SIGACT News* 32, 4 (December 2001), 18–25.
- [40] LAMPORT, L. Fast Paxos. Tech. Rep. MSR-TR-2005-112, Microsoft Research, July 2005.
- [41] LAUER, H. C., AND NEEDHAM, R. M. On the Duality of Operating System Structures. *ACM SIGOPS Operating Systems Review (OSR)* 13, 2 (April 1979).
- [42] LEE, E. A. The Problem With Threads. *IEEE Computer* 39, 5 (May 2006).
- [43] LI, P., AND ZDANCEWIC, S. Combining Events and Threads for Scalable Network Services Implementation and Evaluation of Monadic, Application-Level Concurrency Primitives. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)* (June 2007).
- [44] LIU, B., LIU, P., LI, Y., TSAI, C.-C., DA SILVA, D., AND HUANG, J. When Threads Meet Events: Efficient and Precise Static Race Detection with Origins. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)* (June 2021).
- [45] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'13)* (November 2013).
- [46] NEWELL, A., KLIOT, G., MENACHE, I., GOPALAN, A., AKIYAMA, S., AND SILBERSTEIN, M. Optimizing distributed actor systems for dynamic interactive services. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)* (2016).
- [47] NGO, K., SEN, S., AND LLOYD, W. Tolerating Slowdowns in Replicated State Machines using Copilots. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)* (November 2020).
- [48] ONGARO, D., AND OUSTERHOUT, J. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)* (June 2014).
- [49] OUSTERHOUT, J. Why Threads Are A Bad Idea (for most purposes). In *Presentation given at the 1996 USENIX Annual Technical Conference (USENIX ATC'96)* (January 1996).
- [50] STUTSMAN, R., LEE, C., AND OUSTERHOUT, J. Experience with Rules-Based Programming for Distributed, Concurrent, Fault-Tolerant Code. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)* (July 2015).
- [51] TAN, C., JIN, Z., GUO, C., ZHANG, T., WU, H., DENG, K., BI, D., AND XIANG, D. NetBouncer: Active Device and Link Failure Localization in Data Center Networks. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)* (February 2019).
- [52] VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)* (October 2003).
- [53] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)* (October 2001).
- [54] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)* (June 2015).
- [55] YOO, A., WANG, Y., SINHA, R., MU, S., AND XU, T. Fail-slow fault tolerance needs programming support. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS'21)* (June 2021).
- [56] ZAMORA-GÓMEZ, E., GARCÍA-LÓPEZ, P., AND MONDÉJAR, R. Continuation Complexity: A Callback Hell for Distributed Systems. In *Proceedings of the 21st International Conference on Parallel and Distributed Computing (EuroPar'15)* (August 2015).

- [57] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. K. Building consistent transactions with inconsistent replication. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'15)* (October 2015).
- [58] ZHANG, Q., LIU, V., ZENG, H., AND KRISHNAMURTHY, A. High-Resolution Measurement of Data Center Microbursts. In *Proceedings of the 2017 Internet Measurement Conference (IMC'17)* (November 2017).
- [59] ZHANG, Q., YU, G., GUO, C., DANG, Y., SWANSON, N., YANG, X., YAO, R., , CHINTALAPATI, M., KRISHNAMURTHY, A., AND ANDERSON, T. Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)* (April 2018).
- [60] ZHOU, S., AND MU, S. Fault-Tolerant Replication with Pull-Based Consensus in MongoDB. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)* (April 2021).

A Appendix

A.1 Artifact evaluation

Abstract

A Docker image is provided which contains required dependencies and source code to run the system. Instructions are provided in the `README.md` to reproduce the major results.

Contents

The artifact evaluation includes experiments in Figures 5 and Figures 6. This artifact does not include (1) experiments for comparisons: etcd and ref-copilot, and (2) experiments in B.

Hosting

You can find the publicly available source code at https://github.com/stonysystems/depfast-ae/tree/atc_ae.

Requirements

At least one client plus five servers are used to reproduce the experimental results. Five servers must have an extra disk mounted for slowness experiments. We run all our code on Debian-10, which mainly depends on common Linux libraries (i.e., python, gcc and libyaml-cpp-dev). You can install all dependencies by `bash ./dep.sh`.

A.2 Empowered analysis

DepFast empowers a number of analysis to help programmers understand the fault-tolerance properties of their systems and to detect faults at the run time.

Monitoring with linked coroutines Through the event interface, the DepFast framework can link coroutines together and analyze fail-slow fault propagation. For example, the `RpcEvent` could link the caller and the callee coroutines. The framework will propagate the wait-for information and aggregate them at configured granularity. Figure 7 presents an example of a fail-slow fault propagation graph which shows the wait-for relationship at the node granularity in our DepFast-Raft implementation (§5.1). Each vertex represents a node in a quorum. Each edge is directed and weighted: the direction suggests the wait-for relationship; the weight is the count of the waiting. Each edge is colored. A wait on a potential fail-slow event (e.g., an `RpcEvent`) leads to a *red* edge. A wait on a `QuorumEvent` leads to a *green* edge. This graph is generated and refreshed periodically at runtime. It can be used with graph analysis to detect execution paths which are vulnerable to fail-slow fault, that is, the execution path that contains a red edge. Ideally, this graph should not contain any red edges except for those representing a client issuing requests to a server.

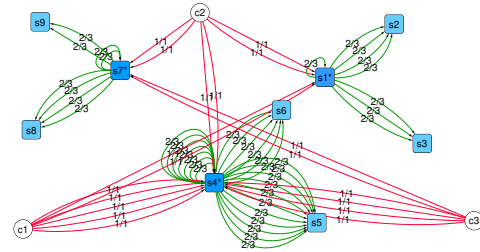


Figure 7: The fail-slow fault propagation graph of DepFast-Raft with three quorums (§5.1). The labels on the edge represents the quorum of the event. “2/3” refers to `QuorumEvent` where 2 responses are needed out of 3 RPCs; “1/1” refers to waiting on a single RPC (clients wait for the leader nodes).

Fault detection. DepFast has a few built-in fail-slow fault detection mechanisms. First, DepFast measures the CPU usage of the worker thread. When the worker thread is awake from `epoll` sleep, it should take up all the CPU core it is running on, because the worker thread does not have thread-blocking calls. The measurement excludes all the `epoll` sleep time and only measures the code executed in the worker thread. If it observes that the worker thread occupies less CPU time than it should, it alerts: either a fault occurs or other programs compete for CPU. Note that this is not perfect detection: the competition could be healthy if it is a shared host; a fail-slow fault could also make the `epoll` sleep longer rather than reducing the CPU utilization.

Second, DepFast measures the time of waiting on each event. If there is a spike for the same event, DepFast will report it. Or, if the wait time repeatedly breaks a user-configured threshold, DepFast will report fail-slow as well.

Third, DepFast exposes the runtime information of resource utilization by allowing applications to register a user-defined detector function. The user-defined detector will be called periodically (taking the monitoring information as inputs) and then make its own decision to notify the application.

B Supplemental evaluation

Figure 8 shows the results of etcd under various fail-slow faults. Our results show that etcd can tolerate these failures well as a production system.

Figure 9(b) and 9(d) shows the throughput and latency CDF of the original Copilot implementation with a fail-slow leader in a 3-replica setup. The experiment verifies that the original Copilot implementation can tolerate a fail-slow node in the cases tested in the Copilot paper. We find that largely due to its immature implementation, in some fail-slow follower cases that are not tested in the original paper, the performance is lower than expected. For example, our experiment of injecting CPU slowness to the original Copilot frequently fails. After some diagnosing, we found that when the follower fails

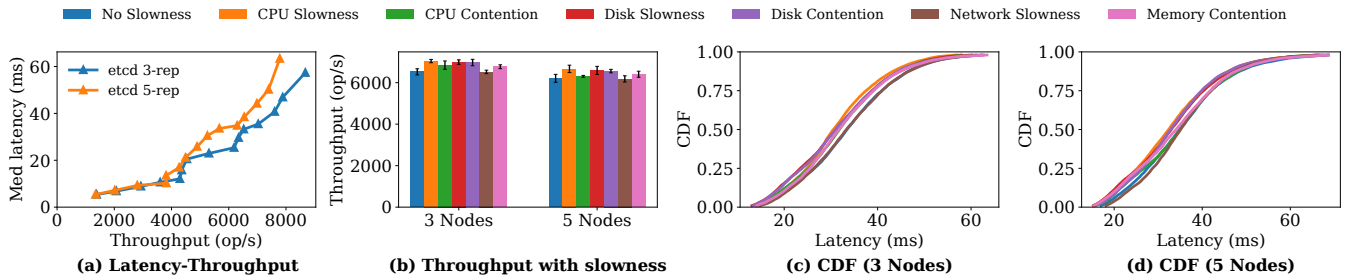


Figure 8: Performance of etcd with various fail-slow faults on follower(s) in the 3- and 5-replica deployments.

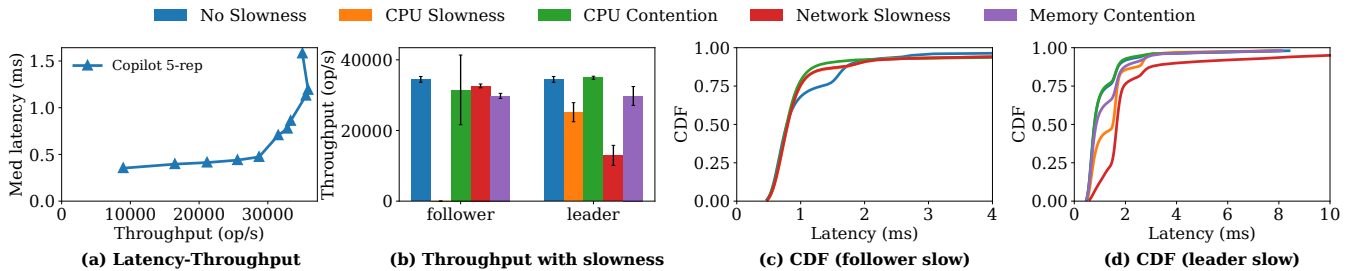


Figure 9: Performance of original Copilot with various fail-slow faults on follower(s) and leader in the 3-replica deployment.

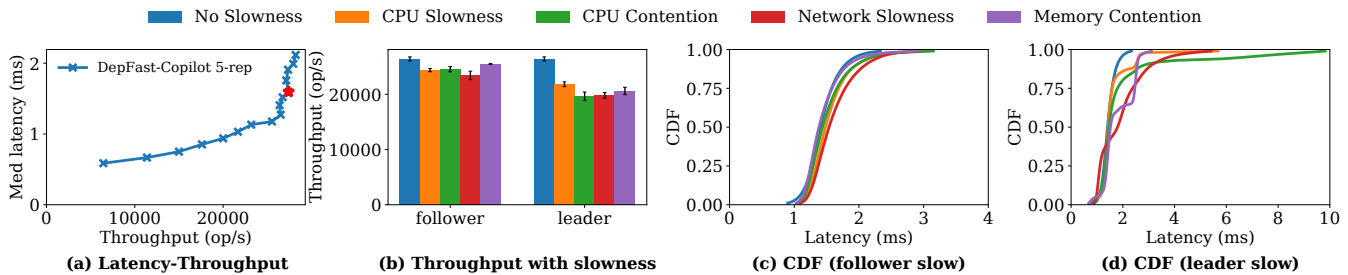


Figure 10: Performance of DepFast-Copilot with various fail-slow faults on follower(s) and leader in a 5-replica setup

slow, it could detect the leader as dead as it cannot process the heartbeats from the leader in time and starts a view change. The leader then steps down and crashes due to what is suspected to be a memory bug. The fail-slow follower will next become the new leader. This results in two live nodes, with one of them failing slow, rendering the cluster into a slow or even stalled state.

Figure 10 shows the evaluation results of DepFast-Copilot in a 5-replica setup, in a similar vein as Figure 6. Figure 10(a) shows the latency-throughput of DepFast-Copilot and the original Copilot implementation. The peak throughput of DepFast-Copilot is 18% higher than the original Copilot.

Figure 10(b)-(d) show the throughput as well as latency CDF of DepFast-Copilot with fail-slow faults injected to follower and leader nodes. Recall from §6.1 that we inject faults on two followers (denoted as “*follower*” in Figure 10) and on one leader and one follow (denoted as “*leader*” in Figure 10). Similar to the results of 3-replica setup, there is no significant downgrade on throughput and latency in terms of both fail-slow follower and leader. For follower slowness, the

decrease in throughput is within 12%. The increase in median and P99 latency are within 10% and 35%, respectively. For leader slowness, the decrease in throughput is within 26%. The latency results are similar to those in a 3-replica setup.

Compared with the 3-replica setup (Figure 6), we find that the 5-replica setup is affected by network and disk I/O spikes more. We discussed the impact of the spikes in §6.2; we elaborate more here. Since we inject one fail-slow node in the 3-replica setup and two fail-slow nodes in the 5-replica setup, the probability of one of the remaining replicas experiencing network or I/O spike is higher in the 5-replica setup than in the 3-replica setup. As every request leads to a quorum read or write, the 5-replica setup is more susceptible to network or I/O spikes. The spikes, in our experience, are common in Azure Cloud (also reported by other studies [16, 51, 58, 59]). Note that Azure uses virtual hard drives that are accessed remotely over the network [59]. As a result, disk writes can also be impacted by network spikes.



High Throughput Replication with Integrated Membership Management^{*}

Pedro Fouto, Nuno Preguiça, João Leitão
NOVA LINCS & NOVA University Lisbon

Abstract

This paper introduces ChainPaxos, a new distributed consensus algorithm for high throughput replication. ChainPaxos organizes nodes in a chain, allowing for a pipeline communication pattern that maximizes throughput, by minimizing the number of messages transmitted. While other proposals have explored such patterns, ChainPaxos is the first that can execute linearizable reads in any replica with no communication overhead, relying only on information used to process updates. These techniques build on a fully specified integrated membership management solution, allowing ChainPaxos's fault-tolerance to be independent of an external coordination service, often used in other solutions, which can lead to possible safety violations in the presence of network partitions.

Our evaluation shows that, when compared with alternative Paxos variants, ChainPaxos exhibits significantly higher throughput and scalability with negligible latency impact. Compared to other solutions with similar communication patterns, besides avoiding the costs of an external coordination service, ChainPaxos's high throughput tends to increase with the ratio of read-only operations.

1 Introduction

Fault-tolerance is a key property for distributed systems, being fundamental to guarantee that they continue to operate despite failures of individual components. To achieve this, the state of the system needs to be replicated over multiple nodes.

A particularly interesting way of providing fault-tolerance is the state machine replication (SMR) [20, 30] approach, which allows to replicate any service providing strong consistency. SMR is achieved by executing the same sequence of deterministic operations on all replicas, making them transition through the same sequence of states.

^{*}This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) under the projects NG-STORAGE (PTDC/CCI-INF/32038/2017) and NOVA LINCS (grant UIDB/04516/2020).

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see <https://www.grid5000.fr/>).

The Paxos [15, 32] consensus protocol and its variants [6, 21, 24, 26–28] have been used as a fundamental building block for implementing SMR, by enabling replicas to agree on the order in which operations are executed. Many practical systems, such as coordination systems, scale-out, in-memory lock services and in-memory databases rely on the performance of their underlying SMR implementation, making it extremely relevant to improve the performance of consensus (or agreement) protocols.

This paper describes the design and implementation of ChainPaxos, a new consensus algorithm for high throughput replication of (deterministic) services. Our goal is to minimize the communication cost of the protocol to achieve the highest possible throughput, both for read and write operations. We achieve this by using a set of complementary techniques. For write performance, we rely on an efficient pipelined communication pattern between replicas, which has been explored and shown effective by previous approaches, notably ChainReplication [33]. This pattern allows to minimize and distribute the number of messages propagated (and therefore processed) by each node to achieve consensus, which highly contributes to maximizing the throughput of write operations. For read operations, we propose a novel scheme for linearizable reads served by a single replica, without incurring in additional communication cost (albeit at the cost of a small increase in latency), which further minimizes the communication overhead of ChainPaxos and increase its throughput.

Contrary to many recent proposals, ChainPaxos does not outsource membership management to an external coordination service (e.g., Zookeeper [12]). Instead, our system features its own integrated membership management solution that allows for the continuous execution of operations during reconfigurations, while uncoupling our system's fault-tolerance from that of an external service. As such, increasing the number of replicas in ChainPaxos effectively increases the maximum number of faults that are tolerated. On the other hand, when leveraging an external coordination service, the fault-tolerance of the replicated systems depends on the fault-tolerance of that coordination service. Additionally, as shown

recently [2], relying on an external coordination service is far from trivial as it makes a system more vulnerable to network partitions, requiring additional logic to ensure correctness.

Our design builds on the insight that it is possible to combine multiple Multi-Paxos messages and exploit a pipeline communication pattern. While this insight is not novel [19], ChainPaxos is, to the best of our knowledge, the first protocol that not only takes advantage of this insight, but also specifies an integrated membership management solution, allowing for lightweight linearizable reads to be served by any replica.

We conduct an extensive experimental evaluation of ChainPaxos where we evaluate its performance against several state-of-the-art variants of Paxos, in particular Multi-Paxos, Egalitarian Paxos, (U)-Ring Paxos, and Chain Replication. The results show that our algorithm provides higher throughput and scalability when compared with other Paxos variants, with even higher gains as both the number of the replicas of the system and the size of operations increase. When compared with other solutions that employ pipeline communication, ChainPaxos shows similar performance for writes, but improved scalability as the ratio of reads increases.

In summary, this paper makes the following main contributions: **i)** a new consensus algorithm that provides high throughput, with integrated membership management that makes it independent from external coordination services (Section 3); **ii)** a novel approach to provide linearizable read operations that distribute the load among all replicas without incurring additional communication costs (Section 4); and **iii)** an extensive experimental evaluation, showing that ChainPaxos provides better performance than state-of-the-art alternatives (Section 5).

2 Related Work

Paxos [15,32] and its variants [6,21,24,26–28] have been used in the design of replicated systems, employing diverse techniques to optimize performance aspects, such as *minimizing latency*, *reducing communication cost*, *distributing the load*, and supporting *linearizable reads*. We now discuss the most popular variants of Paxos, along with Chain Replication [33], an alternative SMR algorithm.

2.1 Minimizing latency

Multiple Paxos variants try to optimize latency. In FastPaxos [18], clients send Accept messages directly to acceptors, skipping the leader. Generalized Paxos [17] extends FastPaxos by allowing non-interfering requests to execute in different orders. In both cases, collisions in client requests result in additional round trips, hindering performance. Flexible Paxos [11] uses different quorum sizes for executing operations (akin to read-write quorum systems [34]), reducing the size of accept quorums and decreasing the latency of accepting operations in the fast path. While ChainPaxos only matches the fast path latency of these protocols when configured to tolerate a single fault, it requires each replica to

handle only one message per operation. In contrast, the $O(n)$ message complexity of Paxos leader and learners in these solutions results in lower throughput.

2.2 Communication cost

Some variants employ chain (or ring) topologies to decrease communication cost. Ring Paxos [28] sends *Accept* messages to all replicas using IP-multicast, with responses being propagated through a ring. IP-multicast limits the operation of the protocol across data centers and negatively impacts the performance under high load when messages are lost. Chain Replication [33] is an SMR algorithm, developed for synchronous systems, where replicas are organized in a *chain* and write operations are forwarded from the head to the tail, with acknowledge messages travelling the opposite way. This approach has the advantage that all replicas send and receive the same number of messages for executing an update. Similarly, U-Ring Paxos [13] propagates messages in a ring topology, with acknowledge messages being forwarded from the tail to the head. These solutions require an external coordination service (e.g., Zookeeper [12]) to reconfigure the system when faults occur, leading to higher operational cost, slower fault-handling, potentially lower fault-tolerance (dependent on that of the external service), and vulnerability to network partitions [2]. ChainPaxos, while having a similar communication pattern, further reduces the number of messages processed by each replica, while handling reconfigurations and faults in an integrated and efficient way.

2.3 Distributing the load

Other variants of Paxos try to distribute the load across replicas. Mencius [22] pre-assigns the leader of each instance to a different node. While providing better throughput, the overall availability suffers since the failure of *any* replica will cause the system to stop until another replica takes over. In Egalitarian Paxos (EPaxos) [26], any replica can commit operations and non-conflicting operations execute in different orders. When there is no conflict, operations commit in a single communication round. Multi-Ring Paxos [24] (based on Ring Paxos) uses a similar approach, while taking advantage of the ring topology to minimize communication. When concurrent operations conflict (which is often the case in SMR), these protocols require extra rounds of communication. Atlas [8] improves on this by allowing some conflicting operations to execute in a single round. Despite trying to distribute the load across all replicas, these protocols still require nodes to send and receive $O(n)$ messages. In contrast, ChainPaxos minimizes the overall load imposed by the protocol, having $O(1)$ message complexity, while also distributing the load across replicas.

2.4 Linearizable reads

In replicated systems, where reads are more frequent than writes, it is important to reduce the cost of read operations to improve overall performance.

Synchronous systems. Chain Replication [33] proposes to execute linearizable reads by contacting a single node: the tail of the chain. When the tail fails, as detected by an external coordination service, clients fallback to the previous node in the chain to continue reading the system state. This solution, however, was designed for a synchronous model where failures can be reliably detected. In an asynchronous system, linearizability can be violated, as the tail can become isolated and be excluded from the chain without knowing, while still serving (outdated) reads. To avoid this, for each read, either the tail or the client would need to contact the external coordination service to verify the current configuration of the chain, which is too expensive. In [33], the authors mention that the coordination mechanism needs to stop clients during reconfigurations, which is unfeasible under network partitions.

Asynchronous systems. Due to a similar reason, solutions based on Paxos cannot execute read operations by contacting only the leader, usually requiring to run a consensus instance for ordering read operations or, in special cases, to contact a quorum of replicas. However, some alternative read schemes to improve replication performance have been proposed. In Smarter [4], reads execute on a single replica, but require a special *whats_my_view* message to be sent to all replicas to gather a majority of replies confirming that no reconfiguration took place concurrently with the read operation. In [10], reads are executed on a single replica, however at the cost of requiring writes to execute in two phases. CRAQ [31] improves reads in Chain Replication by allowing to read from any replica in an asynchronous model, however it only provides per-object linearizability, and for SMR it would require all reads to contact the tail whenever there is a write executing.

In contrast with all these solutions, ChainPaxos includes a novel technique to execute linearizable reads on a single replica, in an asynchronous environment, without ever requiring any additional communication costs. Furthermore, it allows any replica to process reads, thus distributing the read load across all replicas.

3 ChainPaxos

We assume an asynchronous distributed system with n nodes, connected by a network that can lose, duplicate, and deliver messages out of order. Nodes communicate by exchanging messages over a network with a fair loss model that allows the creation of FIFO channels between any pair of nodes. Nodes can fail by crashing, where they stop sending messages.

We follow the SMR model [30], in which each replica holds a copy of the system state and there exists a set of deterministic operations that may output a reply. Replicas start in the same initial state and apply the same sequence of operations, thus guaranteeing that all replicas transition through the same sequence of states and output the same results. We defer the processing of read operations to Section 4.

ChainPaxos is used to order the execution of operations.

The system state includes the application state and the membership of the system, with `AddNode(n)` and `RemoveNode(n)` operations, respectively, adding and removing node n to the replica-set. These operations execute in the state machine, as other application operations, potentially impacting the quorum size of following operations. For correctness, a node can only decide a given instance strictly after knowing the decision of all previous instances (and the current membership).

3.1 Overview

This section introduces ChainPaxos, revisiting Multi-Paxos and Chain Replication to better contextualize our design.

In Multi-Paxos [15, 32], a distinguished proposer, known as *leader*, prepares multiple Paxos instances in a single step (Phase 1), followed by multiple sequential executions of Phase 2 of Paxos. In a fault-free run (Figure 1), the leader sends an *accept* message to all replicas, with each replying to all replicas with an *accept ack* message. Any replica that receives *accept ack* messages from a majority of replicas can decide and execute the request (with the replica that received the operation replying back to the client). With n replicas, the message complexity of the protocol is $O(n^2)$: each replica incurs in $O(n)$ message overhead (the leader sends/receives $2n$ messages). The reply to the client is produced after 2 communication steps between replicas. Alternatively, a replica could send the *accept ack* message only to the leader, which would then forward the decision to all replicas. In that case, the overhead of non-leader replicas decreases to $O(1)$ at the cost of an additional communication step.

Chain Replication [33] leverages a chain topology, forwarding operations from the head to the tail (Figure 2). The tail replies to clients after executing an operation, and sends *ack* messages backwards, to allow replicas to perform garbage collection. In a fault-free run, each replica incurs in $O(1)$ message overhead, with a reply being produced after $O(n)$ communication steps.

The main goals of ChainPaxos's design are: (i) minimize the number of messages each node processes in fault-free runs and make the load uniform, maximizing throughput; and (ii) integrate an efficient fault handling scheme into the algorithm, by taking advantage of Paxos messages, avoiding the need to rely on an external service. To achieve these goals, we leverage the chain topology to combine and forward multiple Multi-Paxos messages in a single ChainPaxos message. As ChainPaxos builds on Multi-Paxos, leader faults can be handled simply by falling back to the first phase of Paxos.

In ChainPaxos, in a fault-free run (Figure 3), the leader sends the *accept* message, including its *accept ack*, to the following replica in the chain. Upon receiving an *accept* message, a replica forwards the message modified to include its own *accept ack*. When the *accept* message reaches the tail of the chain, it sends a message directly to the head with the *accept ack* of all replicas, guaranteeing that the head learns about the decided value. Additionally, it is necessary to in-

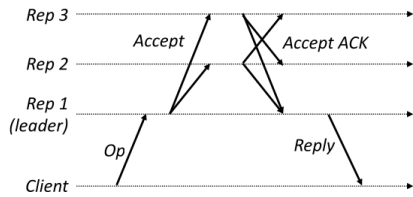


Figure 1: Multi-Paxos message flow on a fault-free run.

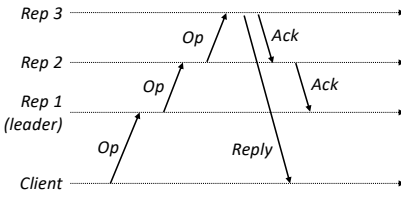


Figure 2: Chain Replication message flow on a fault-free run.

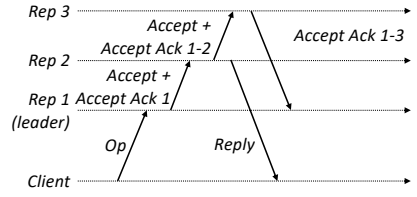


Figure 3: ChainPaxos message flow on a fault-free run.

form the replicas that have not received enough *accept ack* messages to decide the value of the instance – ChainPaxos piggybacks this information in the next *accept* message.

When an *accept* reaches the replica at the middle of the chain, it includes *accept ack* from a majority quorum. Thus, the replica knows that the received request has been decided, and can execute the request and return the result to the client. In the example of Figure 3, with three replicas, the leader and replica 2 form a quorum, with replica 2 replying to the client.

The message flow for fault-free runs achieves the first goal of minimizing the number of messages handled by each replica and keeping the load uniform: a single message is sent and received by every replica. As ChainPaxos is just using a different communication pattern to convey the messages of Multi-Paxos, it can fall back to the regular two phases of Paxos to handle faults. This is the base for achieving the second goal of integrating fault handling in the protocol.

ChainPaxos builds on these ideas to provide high throughput replication by addressing the following challenges: *i*) optimize fault-handling and integrate membership management by leveraging information about the chain topology, thus avoiding the common vulnerabilities/complexity encountered in systems that rely on external coordination services in the presence of network partitions [2]; *ii*) support efficient garbage collection of the information about decided values, which is a common challenge in many variants of Paxos, rarely addressed in the specification of algorithms; and *iii*) integrate a novel mechanism that leverages the chain topology to enable efficient linearizable read operations handled by a single replica without additional communication.

Next, we detail the operation of ChainPaxos, describing the state maintained by each replica and the operation of the protocol in fault-free runs and during reconfigurations. We present correctness arguments for our solution in Annex A.

3.2 Protocol State

Algorithm 1 presents the state of each replica. The first variable group is related with the organization of the system and includes: the members and their order in the chain (*chain*); the identity of the local node (*self*); the next node in the chain that is not marked for removal (*c_{nextok}*); the currently supported leader (*c_{sleader}*); and the replicas for which a *RemoveNode* has been received but not yet decided (*marked*).

Algorithm 1 State of ChainPaxos nodes.

<i>chain</i> : array of nodes	
<i>self</i> : node	▷ local node identifier
<i>c_{nextok}</i> : node	▷ next (unmarked) node in the chain
<i>c_{sleader}</i> : node	▷ supported chain leader
<i>marked</i> : set of node	▷ nodes marked for removal (init : ∅)
<i>np_{leader}</i> : int	▷ special prepare number of the leader
<i>inst</i> : map int × PaxosInst	▷ PaxosInst : (n _a , val, n _{accepts} , decided)
<i>submitted</i> : set of requests	▷ requests submitted by the client
<i>pending</i> : set of requests	▷ requests waiting to execute (leader only)
<i>max_{ack}</i> : int	▷ highest instance acknowledged
<i>max_{accept}</i> : int	▷ highest leader initiated instance (leader only)
<i>amLeader</i> : bool	▷ true if current leader

The second group maintains the information to run Paxos instances. This includes the prepare number (*np_{leader}*) that the leader can use for bypassing the first phase of Paxos. Each replica also maintains a map (*inst*) with the information of Paxos instances including, for each instance, the highest prepare number (*n_a*) used by a leader to accept a value (*val*), the number of nodes that accepted *val* with *n_a* (*n_{accepts}*), and a boolean indicating if the instance was decided (*decided*).

The third group is used for managing client requests. It consists of two sets: *submitted* stores requests received from clients and not yet decided, and *pending* contains the requests received by the leader (redirected from itself or other replicas) but not yet submitted for ordering.

The final group of variables is used for clarity of presentation and stores information that could be derived from other variables, including the highest instance started by the leader (*max_{accept}*), and the highest instance known to have a decided value accepted by all nodes (*max_{ack}*). Each node also keeps track of whether it is the current leader in *amLeader*.

3.3 Fault-free execution

Algorithm 2 presents the ChainPaxos algorithm, with auxiliary functions detailed in Algorithm 3. The highlighted lines represent the logic used in faulty scenarios that require reconfiguration, which are detailed in the next section.

Requests from clients can be received by any replica, and are redirected to the leader (Alg. 2, line 1), which stores them in a set of pending requests (Alg. 2, line 5). The leader, upon receiving a new request, starts a new instance by increasing

Algorithm 2 ChainPaxos algorithm: message flow.

```
1: upon receive <NEW_REQUEST, req> from client do:
2:   submitted ← submitted ∪ {req}
3:   SEND(c_sleader, <REDIRECT_REQUEST, req>)
4:
5: upon receive <REDIRECT_REQUEST, req> from r do:
6:   if self = c_sleader then           ▷ Even if there is no quorum yet
7:     pending ← pending ∪ {req}
8:
9: function STARTINSTANCE
10:  max_apt ← max_apt + 1
11:  SEND(self, <ACCEPT, max_apt, self, np_leader, pending, 0, max_ack>)
12:  pending ← 0
13: upon receive <ACCEPT, n_i, ldr, n_a, val, n_apt_s, m_ack> from r do:
14:   if np_leader ≤ n_a then           ▷ Has not seen higher prepare
15:     UPDATELEADERINFO(ldr, n_a)     ▷ If a prepare was missed
16:     if ¬inst[n_i] ∨ inst[n_i].n_a < n_a then
17:       inst[n_i] ← (n_a, val, n_apt_s + 1, false)
18:     else                             ▷ Repeated accept
19:       inst[n_i].n_apt_s ← MAX(n_apt_s + 1, inst[n_i].n_apt_s)
20:     if inst[n_i].val = RemoveNode(node) then
21:       MARKFORREMOVAL(n_i, node)
22:     if ISQUORUM(n_apt_s) ∧ ¬inst[n_i].decided then
23:       DECIDE(n_i)
24:       DECIDEANDGCUPTo(m_ack)
25:       FORWARD(n_i)
26:
27: upon receive <ACCEPT_ACK, n_i> from r do:
28:   DECIDEANDGCUPTo(n_i)
29:
```

the instance number and generating a new *accept* message (Alg. 2, line 9). The *accept* message contains the following information: (i) the instance number, which the leader tracks in max_apt ; (ii) the id of the leader; (iii) the prepare number, np_leader , used by the leader in its previous prepare message; (iv) the client request (i.e., operation); (v) the number of nodes which have accepted the value (n_apt_s), initialized to 0; and (vi) the highest instance for which the decided value is known to have been accepted by all replicas (max_ack).

The leader is the first to handle the *accept* message of each instance, as it starts a new instance by sending the *accept* to itself (Alg. 2, line 11). Upon receiving an *accept* message for an instance (Alg. 2, line 13), a node stores the information for the instance, increasing the value of n_apt_s to indicate the node itself is accepting the value. If n_apt_s is greater than $n/2$, the message has already been accepted by a majority of nodes, and its value can be decided (Alg. 2, line 22). Otherwise, the value will be decided (and garbage-collected) when an *accept* message is received with m_ack greater or equal to its instance number. This is performed in function `DecideAndGCUPTo` (called in Alg. 2, line 24 and defined in Alg. 3, line 35). This function traverses every (non-garbage-collected) instance up to instance max_ack , marking them as decided (if they were not yet), and garbage-collecting the information about them after their execution. This is safe since all instances up to max_ack have been accepted by every node in the chain.

Finally, the node forwards the *accept* message (with the incremented n_apt_s) to the next node in the chain. If the replica

is the last node in the chain, it sends an *accept ack* message to the leader, signalling that every node in the chain has seen and accepted the instance. Upon receiving this message, the leader executes `DecideAndGCUPTo`, increasing its max_ack which leads subsequent *accept* messages to trigger `DecideAndGCUPTo` in every node across the chain.

The nodes in the second half of the chain (starting from the $n/2^{\text{th}}$ node) can decide instances as soon as they receive the *accept* message, while the first $n/2$ nodes only decide (and execute an operation) after receiving an acknowledgement (the leader via an *accept ack* message, and the other nodes via the max_ack value piggybacked in subsequent *accept* messages).

In a fault-free run, our protocol simply encodes the messages of Multi-Paxos in ChainPaxos messages. A ChainPaxos *accept* message sent by node n encodes the Multi-Paxos *accept* message and the *accept ack* messages of n and all nodes that precede it in the chain. It also encodes the *accept ack* messages of all nodes in the chain for all instances up to m_ack . A ChainPaxos *accept ack* message encodes the Multi-Paxos *accept ack* messages of all nodes in the chain.

3.4 Dealing With Faults and Reconfigurations

To describe how faults and membership reconfigurations are handled in ChainPaxos, we begin by describing the mechanisms used by replicas to suspect other nodes (i.e., fault detection) and then discuss the steps taken by ChainPaxos to reconfigure the system, either keeping the current leader or when the leader is suspected. The main challenge faced by ChainPaxos is that, when using a chain topology, the failure of a single node leads the chain to break, making it impossible for messages to keep flowing along the chain, resulting in a system halt.

Fault Detection: We have implemented two mechanisms for fault suspicion. To pinpoint faults in the chain, each replica expects to receive periodic keep-alive messages from the following node in the chain. If a node does not receive the keep-alive for a configurable period of time, it suspects the node, and requests the leader to remove it, triggering a *Reconfiguration not involving the leader*. In case the tail suspects the failure of the leader (which is its next node), it starts the process of taking leadership (Phase 1 of Paxos), and then starts the process of removing it. This effectively triggers a *Reconfiguration involving the leader*.

We note that, as we assume an asynchronous system, suspecting a node does not necessarily mean that it failed, but rather that there is a chance it might have, as it can just be temporarily slow [29]. However, since a single failed (or just slow) node can block progress in the whole chain, the keep-alive mechanism is important to allow quick removal of suspected nodes, minimizing their negative impact on the overall throughput of the chain. Incorrectly removed replicas can later rejoin the system.

The second mechanism is based on the continuous flow of *accept* messages. If a replica does not receive an *accept*

Algorithm 3 ChainPaxos algorithm: auxiliary functions.

```

1: function MARKFORREMOVAL( $n_i, node$ )
2:    $marked \leftarrow marked \cup \{node\}$ 
3:   if  $node = c_{nextok}$  then  $\triangleright$  We marked the closest unmarked node
4:      $c_{nextok} = NEXTNODENOTMARKED(self, marked)$ 
5:     for  $n \leftarrow max_{ack} + 1, n_i - 1$  do  $\triangleright$  Re-propagate accepts
6:       FORWARD( $n$ )
7: function FORWARD( $n_i$ )
8:   if  $c_{nextok} = leader$  then
9:     SEND( $c_{nextok}, <ACCEPT\_ACK, n_i>$ )
10:  else
11:    SEND( $c_{nextok}, <ACCEPT, leader, n_i, inst[n_i].n_a,$ 
         $inst[n_i].val, inst[n_i].n_{acpts}, max_{ack}>$ )
12: function UPDATELEADERINFO( $leader, n_p$ )
13:   if  $n_{p_{leader}} < n_p$  then
14:      $am_{leader} \leftarrow false$ 
15:      $pending \leftarrow \emptyset$ 
16:      $c_{s_{leader}} \leftarrow leader$   $\triangleright$  Set new leader
17:      $n_{p_{leader}} \leftarrow n_p$   $\triangleright$  Set the prepare number for the leader
18:     for  $req \in submitted$  do  $\triangleright$  Redirect requests to new leader
19:       SEND( $c_{s_{leader}}, <REDIRECT\_REQUEST, req>$ )
20:      $marked \leftarrow \{\}$ 
21:      $c_{nextok} = NEXTNODENOTMARKED(self, marked)$ 
22: function DECIDE( $n_i$ )
23:    $inst[n_i].decided \leftarrow true$ 
24:   if  $inst[n_i].val = RemoveNode(node)$  then
25:      $marked \leftarrow marked \setminus \{node\}$ 
26:      $chain \leftarrow chain \setminus \{node\}$ 
27:   else if  $inst[n_i].val = AddNode(node)$  then
28:      $chain \leftarrow chain \cup \{node\}$ 
29:      $c_{nextok} = NEXTNODENOTMARKED(self, marked)$ 
30:     if  $c_{nextok} = node$  then  $\triangleright$  Was added right next to me
31:       STATETRANSFER( $c_{nextok}, n_i$ )
32:   else
33:     SMREXECUTE( $inst[n_i].val$ )
34:      $pending \leftarrow pending \setminus \{inst[n_i].val\}$ 
35: function DECIDEANDGCUPTO( $n_i$ )
36:   for  $i \in inst \wedge i \leq n_i$  do  $\triangleright$  sequential iteration up to  $n_i$ 
37:     if  $\neg i.decided$  then
38:       DECIDE( $n_i$ )
39:      $inst \leftarrow inst \setminus \{i\}$ 
40:    $max_{ack} \leftarrow n_i$ 

```

for a configurable period of time, it assumes that the leader is faulty and attempts to take leadership. If, during this process, the new leader could not establish a connection to some other node (to send them the *prepare* message), it suspects and starts the process of removing them. To make sure this mechanism operates correctly even if the system is subjected to a low load, the leader issues periodic *accept* messages for a special *NoOP* operation if there are no client requests.

Reconfiguration not involving the leader: We now explain how ChainPaxos reconfigures the chain by removing a suspected node that is not the leader.

When the leader is notified that node n is suspected, it starts an instance with $RemoveNode(n)$ operation to remove node n from the chain. When the instance is decided, n is removed from the chain, updating the variables with the local configuration of the chain (*chain* and *marked*).

When a $RemoveNode$ operation is being propagated, two actions need to be taken to guarantee correctness and progress:

i) guarantee that all previous *accept* messages that might have been lost due to the failure of the node are forwarded to the next correct node (to reestablish the flow of those *accept* messages); and *ii)* guarantee that all subsequent *accept* messages are forwarded through the chain despite faulty nodes, until the $RemoveNode$ operation is decided, removing the faulty node, and repairing the chain.

The former is implemented in $MarkForRemoval$, executed when processing an *accept* message for a $RemoveNode$ operation (Alg. 2, line 21). The node to be removed is added to the set of marked nodes (Alg. 3, line 2). If the node to be removed is the next node that was not previously marked, it is possible that it failed to propagate previous messages through the chain. Thus, the node sends to the next non-marked node any *accept* messages (or *accept ack* for the leader) for instances that have not yet been garbage collected (i.e., instances from max_{ack} to $n_i - 1$) (Alg. 3, line 5). This guarantees that, when healing the chain by bypassing faulty nodes, all *accept* messages will be received by all nodes that will not be removed from the chain, somewhat falling back to the pattern of Multi-Paxos¹.

The latter guarantee is provided by the $Forward$ function (Alg. 3, line 7). This function forwards the *accept* message for a given instance to the next node. When one or more of the following nodes are marked to be removed (because a $RemoveNode$ operation has been received, but has not yet been decided), the function forwards the *accept* message to the next non-marked node. This guarantees that a node that is to be removed in instance n_i will not vote for instances $n > n_i$.

When a leader change occurs while a $RemoveNode$ operation for a node r is being propagated through the chain, it is possible that the operation, while observed by a minority of replicas (that add r to their marked set), is not decided. Following the regular behaviour of Multi-Paxos, the new leader might issue a different operation for that instance. Such operations should be sent to r to ensure correctness. To do so, when a replica learns about the new leader it removes all nodes from the *marked* set and updates the c_{nextok} variable, ensuring that messages flow across all nodes. (Alg. 3, line 12).

Reconfiguration involving the leader: ChainPaxos supports changing the leader by having a node become the leader at a given instance for that and all following instances by executing the first phase of Paxos.

This process is initiated in function $TryToBecomeLeader$ (Alg. 4, line 1). The node selects a prepare number higher than any prepare number already seen in any instance, and sends a *prepare* message for instance $max_{ack} + 1$ directly to all nodes. Although this prepare is for a given instance, it will make the node leader of all instances from that point onward – thus, the prepare number must be larger than any

¹Note that this might lead to nodes receiving multiple *accept* messages for the same instance with the same prepare number from different nodes. This is addressed by considering the highest observed number of acks reported in these messages. This is safe because the forward process employed during recovery never generates cycles.

Algorithm 4 ChainPaxos algorithm: leader election.

```
1: function TRYTOBECOMELEADER
2:    $n_p = \text{NEXTPREPARENUM}(n_{p_{\text{leader}}})$ 
3:    $\text{SEND}(\forall n \in \text{chain}, \langle \text{PREPARE}, \text{max}_{\text{ack}} + 1, n_p \rangle)$ 
4:   upon receive  $\langle \text{PREPARE}, n_i, n_p \rangle$  from  $r$  do:
5:     if  $n_{p_{\text{leader}}} \leq n_p$  then  $\triangleright$  Has not seen higher prepare
6:        $\text{UPDATELEADERINFO}(\text{leader}, n_a)$   $\triangleright$  New accepted leader
7:        $\text{insts}_{\text{accepted}} = \text{GETACCEPTEDINSTSFROM}(n_i)$ 
8:        $\text{SEND}(\text{node}, \langle \text{PREPARE\_OK}, n_i, n_p, \text{insts}_{\text{accepted}} \rangle)$ 
9:
10:  upon receive  $\langle \text{PREPARE\_OK}, n_i, n_p, \text{insts}_{\text{accepted}} \rangle$  from  $\text{rep}$  do:
11:    if  $n_{p_{\text{leader}}} \leq n_p$  then  $\triangleright$  Has not seen higher prepare
12:       $\text{REGISTERPREPAREOK}(n_i, n_p, \text{insts}_{\text{accepted}})$ 
13:      if  $\text{HASPREPAREOKQUORUM}(n_i)$  then  $\triangleright$  Became leader
14:         $\text{amLeader} \leftarrow \text{true}$   $\triangleright$  Can now start new instances
15:        for  $(a_{n_i}, a_{n_a}, a_{\text{val}}) \in \text{ACCEPTEDINSTSFROM}(n_i, n_p)$  do
16:           $\text{SEND}(\text{self}, \langle \text{ACCEPT}, a_{n_i}, \text{self}, n_p, \text{req}, 0, \text{max}_{\text{ack}} \rangle)$ 
17:           $\text{max}_{\text{acpt}} \leftarrow a_{n_i}$ 
18:
```

previously used by any replica. We use $\text{max}_{\text{ack}} + 1$, since it guarantees that previous instances have already been accepted by every node, and all messages regarding those instances can be discarded. As such, nodes only need to maintain the single highest prepare number $n_{p_{\text{leader}}}$ ever received (instead of keeping a prepare for each instance). Since *prepare* messages need to have unique prepare numbers, this number includes an identifier of the node which is used to make sure that no two *prepare* messages from different nodes have the same n_p .

A *prepare* message for a given instance is rejected if the node has already seen a higher prepare number for any instance (either on *prepare* or *accept* messages). Otherwise, the usual Paxos logic is executed for this and all higher instances, with the corresponding *prepare ok* message being returned, which includes all previously accepted values (and corresponding prepare numbers) for the instance indicated in the *prepare* message and all following instances (Alg. 4, line 8). This is necessary as a successful prepare also makes the node the leader of all future instances. From this point until a *prepare* with an higher prepare number is received, the sender of the *prepare* message will be set as the supported leader $c_{s_{\text{leader}}}$ and all pending and future client requests will be redirected to it (Alg. 3, line 12).

Upon reception of a quorum of *prepare ok* messages (Alg. 4, line 13), the node considers itself the new leader. It then executes the regular Paxos logic, but for multiple instances: for all instances for which accepted values exist, it uses the value with the highest associated prepare number as its proposal for that instance, and forwards the corresponding *accept* message over the chain. The regular protocol execution then resumes. In Annex A, we discuss in more detail the correctness of leader election and reconfigurations.

Adding a new replica: For adding a node n to the chain, n sends a request to a replica with $\text{AddNode}(n)$ operation as its value. The leader processes this request by starting an instance that is executed as any other instance of ChainPaxos.

When the instance is decided, the node is added to the

tail of the chain updating the local chain configuration (variables *chain* and c_{nextok}). Once the new node is added, it requests the current state (history of operations or snapshot) from another node at the instance in which the operation to add the node was decided. While this state transfers in the background, the new node can already participate in the following instances actively forwarding messages (although it can only locally execute and garbage collect operations after the completion of the state transfer).

4 Local Linearizable Read Operations

In this section, we discuss our proposal to execute read operations. As we mentioned previously, in Chain Replication, due to the use of an external coordination service, reading from the tail does not provide linearizable reads in the presence of network partitions, as the tail might become partitioned and not be aware that it was removed from the system. Guaranteeing linearizable reads requires contacting the external configuration service, which defeats the purpose of the low overhead achieved by only contacting the tail. Due to similar issues, most SMR protocols only support linearizable reads by executing them as normal consensus operations or, in some cases [5], by contacting a quorum of replicas (and falling back to executing the read as a normal operation when conflicts occur). We now discuss how we leverage on the chain topology and our integrated membership management to provide linearizable reads without any added communication cost.

To provide linearizable reads, it is necessary to guarantee that the result of a read reflects a state that, at the moment the read is received, is at least as recent as the most recent state for which any node has returned a result (either for a read or for a write). The base intuition of our proposal is that a node can guarantee this property by waiting for a message to loop around the entire chain, making sure that the local node is as up-to-date as any node was at the moment the message started looping around the chain.

Based on this intuition, our solution for linearizable reads works as follows. Clients issue read operations to any replica in the chain. Upon receiving the operation, the replica locally registers that the operation depends on the lowest unseen consensus instance (but no information is sent to other nodes). For instance, if the highest instance that the replica has seen so far is 6 (regardless of it being decided or not), the read operation will depend on instance 7. Upon receiving the *accept ack* message to the consensus instance for which the operation depends on, the read operation is performed locally in the local committed state and the reply is sent to the client.

This protocol implements linearizable reads by enforcing the following properties: **i)** a read r returns a value that is at least as recent as any value outputted by the protocol at the moment the read was received. By waiting that the following consensus instance is acknowledged and executing the read in the current local state, a replica is assured that the result of any read that was returned at any replica before the reception

of r cannot be more recent than the result that will be returned for r – this follows from the properties of ChainPaxos, which guarantee that as messages loop the chain they make the state of replicas advance, so that the following replica in the chain is in a state that is at least as recent as the previous replica. As such, if some replica has already returned a value for state s_i , by waiting that the following consensus message is acknowledged, which requires a full loop of an operation through the chain, the local replica state will be at least as recent as s_i . Due to the same reason, the result of a read will also reflect the result of any committed write operations, at the moment the read was received; **ii**), upon a reconfiguration, a node that is partitioned from the chain will not return stale values: when a replica loses connection to the others, either it is eventually removed from the chain, preventing it from ever replying to client read operations or it eventually reconnects to the other nodes, allowing it to continue responding to read requests. In the latter case, since the replica was not removed from the chain, no progress was made while it was partitioned, thus linearizability is not lost.

Our proposal trades a potentially higher latency (compared to executing a read as a normal operation) for the possibility of processing a read locally at any node, without additional consensus instances or communication steps. This leads to lower communication and processing overhead, and allows to balance the load of read operations across all replicas, leading to better overall performance. Under low load, write operations may be less frequent, which could delay read operations. We note however, that the head of the chain issues periodic *NoOP* operations if no write is received, as to show to the other replicas that the head is still correct, hence the maximum latency of reads in scenarios with a low load will be controlled by the frequency of these *NoOP* operations. Alternatively, to ensure faster read processing, a replica processing a read which has not received the message for the next consensus instance after some configurable timeout can forward the read to the leader to be executed as a normal operation (which in turn will allow other pending reads to complete).

5 Evaluation

This section reports the experimental evaluation of ChainPaxos in a broad range of scenarios. We start by assessing the performance and scalability in CPU-bound and network-bound settings (Section 5.2), and the impact of our novel read protocol (Section 5.3), using a replicated key-value store application under the YCSB workload [7], when compared with other consensus protocols. Then, we report the results of integrating ChainPaxos with ZooKeeper [12], by replacing the Zab [14] replication protocol (Section 5.4). Finally, we study how ChainPaxos behaves in a geo-replicated setting (Section 5.5) and the impact of reconfigurations in our integrated membership when compared to using an external coordination service (Section 5.6).

We have implemented a prototype of ChainPaxos in Java,

using a framework for building distributed protocols, Babel [9], which relies on the Netty [1] framework for the communications module. Similarly to other authors [8, 26], to guarantee fairness in our comparisons, the other consensus protocols were implemented using the same codebase as ChainPaxos. This guarantees that the results are not influenced by specific implementation aspects, such as the programming language, client communication patterns or differences in optimizations (such as batching). Each protocol was implemented following the description presented in their respective publications as well as available code bases for EPaxos [25] and Ring Paxos [23]. For the latter, as proposed by the authors, we limit the number of concurrent instances the leader can start as a form of flow control to mitigate the loss of multicast messages and include a mechanism for recovering from lost messages.

Each replica includes: an application (either the replicated key-value store or Zookeeper), which receives client requests, submits them for ordering, and replies to the client when the operation is executed; a proxy, serving as the intermediary between the application and the consensus protocol, also redirecting operations to the consensus leader when applicable and; the consensus solution itself, which receives operations from the proxy and notifies it once their ordering is decided.

5.1 Experimental Setup and Parameters

The experiments were conducted on the Grid5000 testbed [3], using a cluster of machines with an Intel Xeon Gold 5220 CPU with 18 cores and 96 GiB DDR4 RAM. Machines are connected through a 25 Gbps Ethernet switched network. Each replica executes in its own machine, and clients (running YCSB [7]) execute on 3 independent machines (with multiple client threads per machine). Each client thread connects to a replica for executing operations in a closed-loop.

Every protocol is executed in similar conditions, with the exception of Chain Replication that uses Zookeeper as the external management service (following [33]). For all protocols, the leader is elected at the start of the experiment and the protocols run multiple consensus instances in parallel. All results are the average of 5 independent runs, discarding the start and end periods of each experiment. In all results presented, the standard deviation between runs is always below 10%.

In addition to ChainPaxos, Chain Replication [33], and Ring Paxos [28], we report as: *EPaxos*, the execution of EPaxos [26] in a workload where all operations conflict (which is the same case of other baselines); and *EPaxos-NoDep*, the execution of EPaxos in a workload where no two operations conflict, which is equivalent to running multiple independent Paxos instance in parallel. We note that this is an unrealistic workload, as it would require all operations to be independent from each other, being presented only to provide the best (theoretical) results for a protocol following the strategy of EPaxos. *MultiPaxos* refers to the variant of Multi-Paxos [16] where acceptors forward their *accept ack*

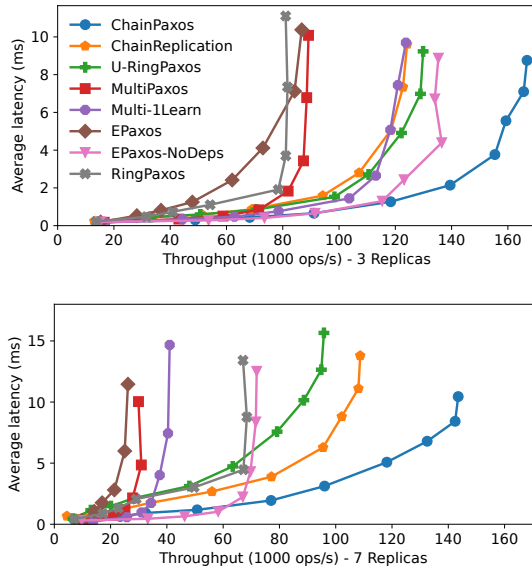


Figure 4: Performance for operations with 128 bytes (CPU bottleneck).

messages to all replicas, whereas *Multi-1Learn* represents the variant where acceptors only send the *accept ack* to the leader that, upon collecting a quorum of replies, issues a *decided* to all replicas - this protocol has a message flow equal to Raft [27] in the normal case. *U-Ring Paxos* [13] is a variation of Ring Paxos, using unicast instead of multicast, with a message flow similar to our solution and Chain Replication.

5.2 Performance in a Single Data Center

This section reports the results obtained in a single data center, running the YCSB benchmark with a replicated key-value store application. We study scenarios that attempt to saturate the CPU and the available bandwidth, by varying the size of the data stored in the key-value store.

CPU Bound. Figure 4 shows the performance of each protocol in a CPU-bound scenario. For this experiment, clients execute small (128 bytes) operations and no batching is employed (i.e., each operation is executed in an individual consensus instance). Clients connect uniformly at random to a replica, and receive a reply after the operation is executed in that replica. While this does not provide optimal latency for some solutions, it maximizes throughput by distributing the load of handling client requests as much as possible.

These results show that, by pipelining a single message per each operation through all replicas, ChainPaxos minimizes CPU usage, achieving the best performance and scalability. Chain Replication and U-RingPaxos perform worse, as they propagate some extra messages: acknowledge messages in the former, and proposals being propagated to the leader through the chain in the latter. These messages could be batched or piggybacked with a small penalty to latency. We note that the throughput of ChainPaxos with 7 replicas, which tolerates 3

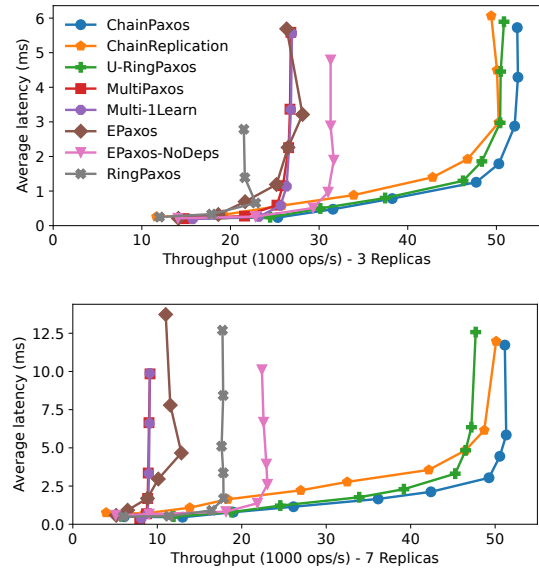


Figure 5: Performance with network bottleneck.

faults, is higher than that of Chain Replication with both 3 and 7 replicas, which tolerate 2 and 6 faults, respectively.

For both versions of MultiPaxos, the leader (and all replicas in regular MultiPaxos) transmits and receives messages from, at least, a majority of replicas, resulting in higher CPU usage and lower performance. The impact of this effect increases with the number of replicas. For EPaxos, when all operations need to be ordered, the algorithm requires two rounds of communication, leading to an higher number of messages and lower throughput. The execution of *EPaxos-NoDeps* is similar to executing multiple parallel MultiPaxos instances, distributing the load among replicas. This leads to an higher throughput than MultiPaxos and EPaxos that, unlike ChainPaxos, also decreases with the number of replicas, as more messages need to be processed. Furthermore, we note that *EPaxos-NoDeps* is not totally ordering all operations, as other protocols do. RingPaxos is tricky to tune, as a single lost multicast message can stall the entire system. Even for our best configuration (with 150 simultaneous consensus instances), RingPaxos performance is worse than U-RingPaxos.

Overall, these results show that lowering the number of messages processed by each replica allows to achieve higher throughput with a negligible latency overhead. Furthermore, the throughput of chain-based protocols degrades very slowly when increasing the number of replicas, while the throughput of other protocols degrades quickly, as the number of messages processed by each node depends on the number of replicas in the system. This is relevant for supporting critical systems with high availability requirements.

Network Bound. Figure 5 presents the performance in a network-bound scenario. For this experiment, the bandwidth of replicas is limited to 1Gbps, with clients issuing 2048 byte operations, saturating the bandwidth of the replicas without

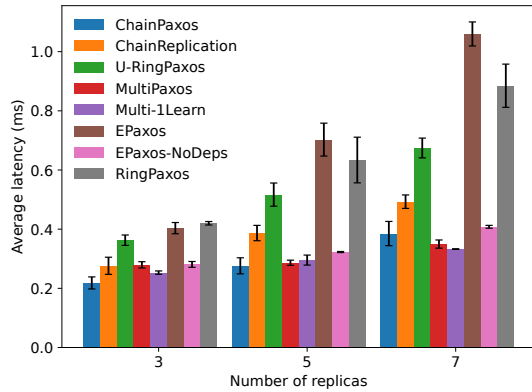


Figure 6: Latency under low load.

satürating their CPU. For saving the bandwidth consumed in redirects and maximizing the bandwidth available for the consensus protocol, all clients connect directly to the leader/head (uniformly distributed in both EPaxos variants).

Results show that ChainPaxos, ChainReplication, and U-RingPaxos, by only receiving and transmitting each operation once, achieve maximum use of available bandwidth. For these solutions, the replicas were consuming approximately 900 Mb/s of both inbound and outbound bandwidth. This allows the system to maintain its performance with an increasing number of replicas. For MultiPaxos, since the leader needs to transmit each operation to all other replicas, its bandwidth usage is disproportionately higher than that of other replicas, limiting their throughput. Furthermore, the throughput decreases with the number of replicas. EPaxos versions suffer from the same issue, but since EPaxos uses multiple leaders, it distributes the load of the leader across all nodes, leading to better scalability than MultiPaxos. *EPaxos-NoDeps* requires less communication steps, having higher throughput, but still far from ChainPaxos. For RingPaxos, the higher message size results in more frequent message losses. Even configuring the number of concurrent instances to 20 as to achieve the best results, the performance is substantially lower than that of ChainPaxos and Chain Replication.

Latency with a fixed throughput. Figure 6 shows the latency with a fixed load – clients execute 9000 operations per second, using payloads of 128 bytes. In this experiment, clients are setup to minimize latency: in RingPaxos and MultiPaxos clients connect directly to the leader; in EPaxos clients connect to all replicas uniformly; in Chain Replication and U-RingPaxos clients connect to the tail; and in ChainPaxos to the replica in the position $n/2 + 1$. Error bars present the standard deviation of the results.

The results show that, with 3 replicas, ChainPaxos and MultiPaxos variants exhibit the lowest latency, since they can respond to client requests after a single communication step. With increasing numbers of replicas, the latency of ChainPaxos increases, while the latency of both MultiPaxos variants remains mostly unaffected. Since both U-Ring Paxos and

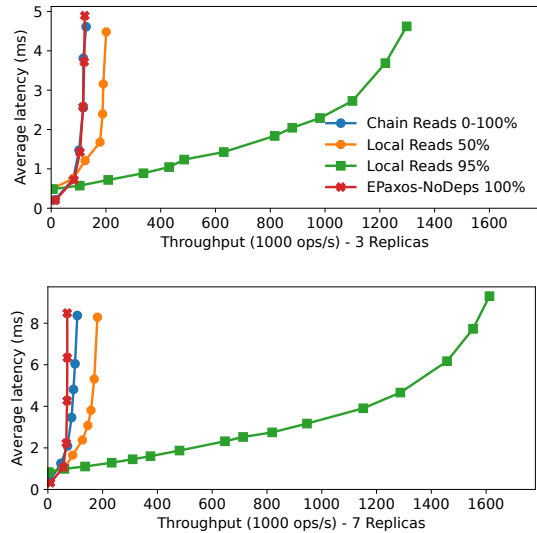


Figure 7: Performance with read operations.

Chain Replication require more communication steps until a reply is generated, their latency is always higher than ChainPaxos. We note that ChainPaxos with 5 replicas presents a similar latency to Chain Replication with 3 replicas, in which case both tolerate the failure of 2 replicas. In EPaxos, since conflicts lead to extra communication rounds, the variant where all operations conflict (*EPaxos*) naturally shows higher latency. For RingPaxos, multicast message drops (which happen even without saturation) and retransmissions lead to higher latency.

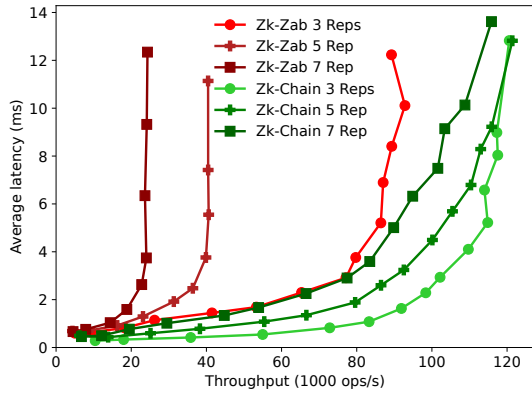
5.3 Performance of Read Operations

Figure 7 shows the impact of ChainPaxos’s novel linearizable read approach in workloads with different read ratios and payloads of 128 bytes. The throughput of executing reads as normal (consensus) operations (*Chain Reads*) is constant, regardless of the ratio of read operations. For our novel approach (*Local Read*), the throughput is much higher than executing reads as normal operations, and the throughput scales both with the ratio of read operations and with the number of replicas. This is explained by the fact that as reads impose no overhead to the consensus protocol and the load is distributed evenly among replicas, more replicas can process more reads. The high throughput of ChainPaxos’s local linearizable reads comes at the cost of a small additional latency under low load.

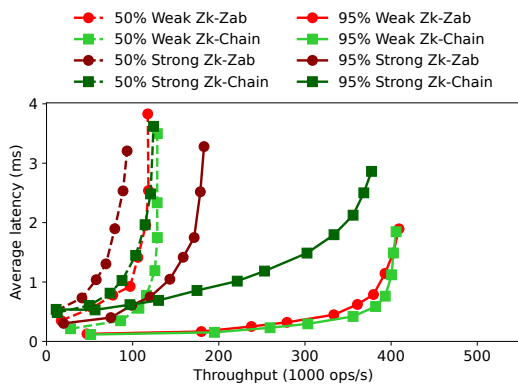
For comparison, we include the results of *EPaxos-NoDeps*, the Paxos-based protocol with best performance in the previous results. The results show that ChainPaxos achieves a significantly higher throughput than *EPaxos-NoDeps*.

5.4 Zookeeper case-study

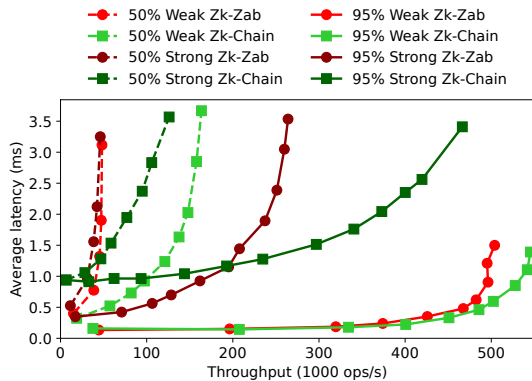
To evaluate the performance of our protocol in a more realistic scenario, we adapted ZooKeeper [12] to use ChainPaxos as its consensus protocol, instead of Zab [14]. While some features were not implemented, such as ephemeral nodes, our implementation fully supports creating, updating, and retrieving



(a) Write-only workload



(b) Mixed workload with 3 replicas



(c) Mixed workload with 7 replicas

Figure 8: Performance of Chain-based Zookeeper vs original Zookeeper.

znodes. We evaluated the performance of our implementation (*ZK-Chain*) against the original ZooKeeper using Zab (*ZK-Zab*), in a setup similar to the CPU bound scenario of Section 5.2. The results are presented in Figure 8.

For a write-only workload (Fig. 8a), the results show that ChainPaxos achieves higher throughput than the original Zookeeper, with the difference increasing with the number of

replicas. This is due to the lower number of messages of our protocol (Zab’s message pattern is similar to *Multi-1Learn*).

Figures 8b and 8c present mixed workloads (50% and 95% of read operations), with both weak and strong reads. *Weak reads* represent the regular reads of ZooKeeper, where a replica replies with its current state, allowing for stale data to be served (e.g., with late replicas and under network partitions). *Strong reads*, in our solution, are executed using linearizable local reads. While ZooKeeper does not support linearizable reads, the authors suggest issuing a *sync* operation before a read as a close approximation of linearizability in most cases. The results show that, unlike with Zab, the strong reads with ChainPaxos scale to a throughput similar to executing weak reads. Overall, the throughput with ChainPaxos is higher than with Zab for the same setting, and the difference increases with the number of replicas.

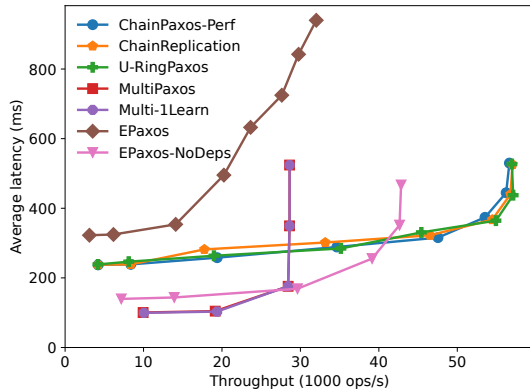
5.5 Performance in a Geo-Replicated Setting

We evaluated our protocol in a geo-replicated setting by emulating an environment with 5 sites. Using the Linux `tc` command, we limited the bandwidth to 1Gbps, and increased latency to the following values, extracted from <https://cloudping.co>, related to AWS EC2 data centers.

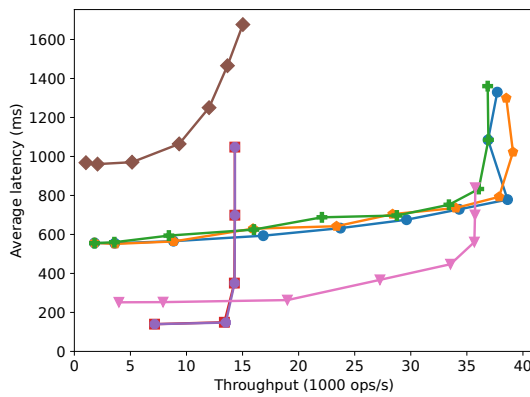
Sites	A	B	C	D	E
North Virginia (A)	-	92	127	204	186
Frankfurt (B)	88	-	210	288	279
São Paulo (C)	122	207	-	338	359
Sydney (D)	211	292	325	-	161
Seoul (E)	188	287	309	156	-

In these experiments, we did not use Ring Paxos, since IP multicast is typically unavailable across data centers. The replica in site A is always the leader/head. Experiments with 3 replicas use sites A, B, and C. Clients connect to the replica that leads to the best performance: the leader for MultiPaxos; evenly distributed for EPaxos, and; the tail for chain-based solutions.

Figure 9 presents the throughput and latency when using all available bandwidth. As within a single data center, ChainPaxos, Chain Replication, and U-Ring Paxos are able to make optimal use of available bandwidth, providing higher throughput than other protocols that order all operations. The EPaxos variant without inter-operation dependencies is able to maintain its throughput with a varying number of replicas, since the cost of transmitting each operation to all nodes is divided among the multiple leaders. However, we remind the reader that this configuration of EPaxos provides weaker guarantees than the other alternatives. As for latency, the latency of the chain-based solutions degrades as the number of replicas increases, as it takes longer for the messages to traverse the chain. For a high number of replicas, MultiPaxos variants provide lower latency, as communication between the leader and other replicas proceeds in parallel, although with, at most, half the throughput of ChainPaxos.



(a) 3 Replicas



(b) 5 Replicas

Figure 9: Performance in geo-replicated setting.

5.6 Impact of Reconfiguration

In our final experiment we evaluate the impact of reconfiguration, comparing ChainPaxos that uses its own integrated management mechanism and Chain Replication that uses an external management scheme based on Zookeeper (executing on dedicated machines). We conduct these experiments in the geo-replicated scenario with independent Zookeeper instances at sites A, B, and D. This distribution minimizes latency for replicas without a local Zookeeper replica. We used 1s timeouts to suspect the failure of another node (both in ChainPaxos and in ZooKeeper).

Experiments run for 90 seconds. Every 10 seconds the following reconfiguration events occur (denoted by vertical red lines for replica failures and green lines for replica additions): 10s) the tail node fails; 30s) the middle node fails; 50s) the head/leader fails; 70s) the head and middle replicas fail simultaneously. Replicas are added at 20s, 40s, 60s, 80s, in sites where a replica had previously failed. Clients issue operations to a random active replica to distribute the load.

Figure 10 shows the throughput observed during the experiments. Despite Chain Replication using additional resources (3 extra machines executing Zookeeper), it takes more time to perform a reconfiguration than ChainPaxos, particularly when

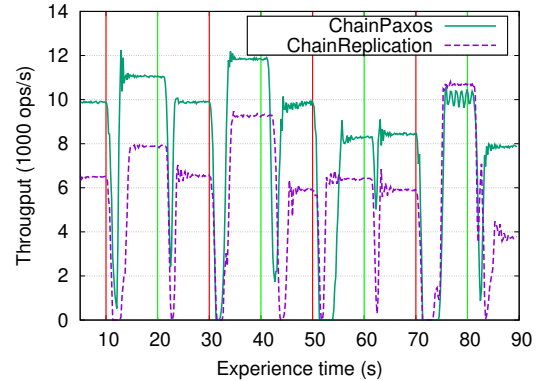


Figure 10: Reconfiguration

adding replicas to the set (green vertical lines). This happens because any reconfiguration has to be coordinated through Zookeeper. However, ChainPaxos takes longer to perform the reconfiguration when the leader fails because it resorts to the regular communication pattern of Paxos, whereas Chain Replication only fetches the new leader from Zookeeper. When the leader and middle nodes fail simultaneously (70s), both solutions take the same time to perform reconfiguration because ChainPaxos can handle both reconfigurations in parallel (albeit using two operations), whereas Chain Replication performs two sequential reconfiguration steps with Zookeeper. In general, ChainPaxos handles reconfiguration faster than Chain Replication without the cost of requiring additional machines to run the external management system, while avoiding the vulnerabilities to network partitions that can compromise the safety of the system [2].

6 Final Remarks

This paper presented ChainPaxos, a distributed consensus algorithm for high throughput replication of deterministic services. ChainPaxos exploits a pipeline communication pattern which allows to reduce the number of messages that each replica needs to send and process, while leveraging the foundations of Paxos to allow leader exchanges. Unlike previous solutions that exploit this communication pattern, ChainPaxos relies on a novel approach to execute linearizable read operations without incurring in any additional communication cost. Finally, ChainPaxos integrates membership management within the protocol. The fully specified algorithm fills an empty space in the literature and, unlike many recent proposals, decouples the fault-tolerance of ChainPaxos from that of the external coordination service. Our extensive evaluation shows that ChainPaxos achieves higher throughput and better scalability when compared to state-of-the-art solutions. Furthermore, our approach for executing linearizable reads has a huge impact on scalability. The results illustrate the benefits of our solution in the context of a key value store and the Zookeeper coordination services (where ChainPaxos leads to better performance than Zab).

References

- [1] Netty framework. <https://netty.io/>.
- [2] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 351–368, 2020.
- [3] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid’5000 testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.
- [4] William J Bolosky, Dexter Bradshaw, Randolph B Haagens, Norbert P Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proc. NSDI’11, USENIX Conference on Networked Systems Design and Implementation*, pages 141–154, 2011.
- [5] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation, OSDI ’99*, page 173–186, USA, 1999. USENIX Association.
- [6] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.
- [7] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.
- [8] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys ’20*, New York, NY, USA, 2020. Association for Computing Machinery.
- [9] Pedro Fouto, Pedro Ákos Costa, Nuno Preguiça, and Joao Leitao. Babel: A framework for developing performant and dependable distributed protocols. *arXiv preprint arXiv:2205.02106*, 2022.
- [10] Rachid Guerraoui, Dejan Kostic, Ron R Levy, and Vivien Quema. A high throughput atomic storage algorithm. In *27th International Conference on Distributed Computing Systems (ICDCS’07)*, pages 19–19. IEEE, 2007.
- [11] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696*, 2016.
- [12] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA, 2010.
- [13] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: High-throughput atomic broadcast. *The Computer Journal*, 60(6):866–882, 2017.
- [14] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE, 2011.
- [15] Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.
- [16] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.
- [17] Leslie Lamport. Generalized consensus and paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.
- [18] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.
- [19] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. Technical report, 2009.
- [20] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, 2010.
- [21] Leslie Lamport and Mike Massa. Cheap paxos. In *International Conference on Dependable Systems and Networks, 2004*, pages 307–314. IEEE, 2004.
- [22] Yanhua Mao, Flavio P Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 369–384. USENIX Association, 2008.

- [23] Parisa Jalili Marandi. U-Ring paxos code. <https://github.com/sambenz/URingPaxos>. (Accessed Oct 2019.).
- [24] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. Multi-ring paxos. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.
- [25] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Epaxos code. <https://github.com/efficient/epaxos>. (Accessed Mar-2019.).
- [26] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13*, pages 358–372, New York, NY, USA, 2013. ACM.
- [27] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.
- [28] Parisa Jalili Marandi, M. Primi, N. Schiper, and F. Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 527–536, June 2010.
- [29] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–14, 2015.
- [30] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.
- [31] Jeff Terrace and Michael J Freedman. Object storage on craq: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference*, 2009.
- [32] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42–1, 2015.
- [33] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6, OSDI'04*, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.
- [34] Avishai Wool. Quorum systems in replicated databases: Science or fiction? *IEEE Data Eng. Bull.*, 21(4):3–11, 1998.

A Correctness

In this section, we present the correctness argument for ChainPaxos, by showing that the execution of an instance of ChainPaxos is equivalent to an execution in Multi-Paxos. We consider the following cases: (i) an instance where all nodes agree on the leader; (ii) an instance that elects a new leader; (iii) instances following a new leader election; and (iv) instances where a replica is removed or added.

A.1 All nodes agree on the leader

This case is the one described in Section 3.3. In Multi-Paxos, an equivalent run would consist in: (i) the leader sending the *accept* message to all replicas; (ii) each node replying with an *accept ack* message to the leader that learns the decided value; (iii) the leader informing all other nodes of the learned value.

In ChainPaxos, the *accept* message is also sent to all nodes, but instead of being sent directly it is sent indirectly as the *accept* message is forwarded across all nodes of the chain.

The *accept* message also encodes the *accept ack* messages of the nodes through which it passes. When the leader receives the *accept ack* message from the tail of the chain, this message is equivalent to having all nodes sending the *accept ack* message to the leader in Multi-Paxos.

Piggybacked in the following *accept* messages (even for the NOOP request), the leader sends to all other nodes information that it has received the *accept ack* from all nodes of the chain (m_{ack} in *accept* message), which is equivalent to forwarding the learned value to all replicas in Multi-Paxos.

As messages are used to update the local state using the same logic in both ChainPaxos and Multi-Paxos, the execution of an instance in ChainPaxos is equivalent to the execution of an instance in Multi-Paxos.

A.2 Instance with a leader election (two-phases)

We start by considering the effects of an instance executed with the two phases of the Paxos protocol. In this case, the first phase of both protocols is identical – ChainPaxos uses the same message flow as Paxos, since *prepare* and *prepare ok* messages are sent directly between the node starting the prepare and all other replicas. The logic to process the *prepare* message in ChainPaxos differs from Multi-Paxos in the following aspect: ChainPaxos only acknowledges *prepare* messages for a given instance with prepare numbers higher than any prepare number used by any other leader in any other instance. The only implication of this is that the replica executing the prepare, might time-out several times. This is not different from Multi-Paxos, where the replica will complete the prepare in the current instance, and then, if any other following instance had already been executed by a different

leader (potentially using the special prepare number zero), it would have to repeat the prepare phase for each such instance. This is true because in ChainPaxos the prepare number is continually used by a leader in all subsequent instances after the one in which it complete the prepare phase successfully.

The second phase of ChainPaxos consists in sending the *accept* message to all nodes. As explained in the previous case, ChainPaxos flow of messages is equivalent to the Multi-Paxos flow of messages. As the logic employed to process the messages is the same as in Multi-Paxos, also in this case, the execution is equivalent to that of an instance of Multi-Paxos.

A.3 Instances following a new leader election

When a new leader is elected, by running the first phase of Multi-Paxos for instance n_i , it becomes the leader of all instances n_j such that $n_j \geq n_i$, for which it will subsequently issue *accept* messages through the chain (in order). To prove the correctness of ChainPaxos, we show that this execution is equivalent to running the two phases of the Paxos protocol for all instances n_j , such that $n_j \geq n_i$.

The processing of the *prepare* message ignores the prepare if the node has seen an higher prepare number employed by another leader (for any instance). If the prepare is not ignored, it runs the Paxos logic for all instances n_j , such that $n_j \geq n_i$, with the *prepare ok* message including the information relative to all these instances. This information includes any accepted value for each of those instances and the prepare number associated with that accepted value. The processing of the *prepare ok* message also executes the Paxos logic for all instance $n_j \geq n_i$, such that the new leader will send new *accept* messages for all those instances (in order) where, if there was already a value accepted by any replica for that instance, the new leader will propose that value and, similar to Multi-Paxos, if more than one value had been accepted by different replicas, the leader proposes the value with the highest associated prepare number. This ensures that for every instance $n_j \geq n_i$, if some value had already been accepted by a majority of replicas, then the new leader will propose that value with his current propose value.

Thus, executing the first phase of the Multi-Paxos in ChainPaxos for instance n_i and issuing the *accept* for all instances n_j , such that $n_j \geq n_i$ as described above is equivalent to executing the two phases of the Paxos protocol for all instances n_j , such that $n_j \geq n_i$, thus ensuring the correctness of ChainPaxos.

A.4 Leader Conflicts

As in Paxos, it is possible that two nodes receive a quorum of *prepare ok* messages for different prepare numbers concurrently. In this case, *accept* messages from the node with the lowest prepare number will be dropped during their propagation and will never be accepted by a majority of replicas.

The processing of *accept* messages also sets the leader and prepare number being used by the leader in all replicas. This is needed because a minority of replicas might have missed the *prepare* message.

If a previous leader is incorrectly suspected of being faulty, it might send *accept* messages for instances $n_i \geq n$ while a new node becomes leader in instance n . In this case, just like in regular Multi-Paxos, there are two possible scenarios for each instance n_i : **i)** the *accept* message of the previous leader reached a majority of nodes (i.e., has been decided) before the *prepare* message of the new leader, in which case at least one *prepareOk* message received by the new leader will contain the value proposed by that *accept* message, and the new leader will simply propose that same value; **ii)** the *accept* has been “cut-off” by the *prepare* message of the new leader (i.e., a node in the chain rejected the *accept* after receiving the *prepare* with an higher sequence number). In this case, the new leader may or may not receive the value proposed by that *accept* in a *prepareOk* message. Regardless, since the instance had not been decided, both alternatives are correct.

A.5 Removal and addition of a replica

We now discuss the correctness of ChainPaxos when reconfiguring the system to either remove or add a replica, which has been presented in Section 3.4. Such reconfigurations resort to special SMR operations which have to be ordered by ChainPaxos and executed by replicas. A challenge that is present in such reconfigurations is that the number of replicas that constitute a majority of the system might change due to the execution of these operations.

In ChainPaxos, we define a minimum quorum size, and then vary the size of the quorum required to decide operations to always be a majority quorum. For instance, assuming a configuration with 5 initial replicas (where a majority quorum is 3), and a minimum quorum size of 3, adding 2 replicas (to a total of 7), would increase the majority quorum to 4. If 2 replicas are then removed, the majority quorum will be back to 3. However, due to the minimum quorum size, removal of further replicas would not decrease the quorum size below 3, which also ensures that the number of replicas in the system can never be below 3. The minimum quorum size is independent from the initial configuration, serving as a threshold below which we do not wish the system to continue functioning.

Note that in ChainPaxos no replica considers an instance as decided before knowing the operations that have been ordered in all previous instances, hence a replica will never use the incorrect quorum size to decide (and execute) an operation.

We start with the addition of a replica. To ensure correctness we need to show that the replica that is added in an instance n_i will be considered towards forming the majority of *accept ack* messages necessary to decide any instance $n_j > n_i$. This derives from the agreement property of Chain-

Paxos, which ensures that all (correct) replicas will agree on the instance in which the new replica is added to the system. Since replicas only consider an operation as decided after learning the decided values for all previous instances, no replica will ignore the participation of the new replica when deciding the value of any instance n_j , since the replica has already been added on instance n_i (i.e., the instance where the new replica is added), affecting the size of the quorums.

Regarding removal of nodes, to ensure correctness we need to show that the removal of a replica in instance n_i , makes it impossible for that replica to affect the decided value in any instance $n_j > n_i$ (i.e., the *accept ack* messages of that replica are never considered to achieve a majority in such instances). As discussed previously, the *accept* message that is forwarded along the chain, in round n_i , to remove a replica r is forwarded by the node directly before r to both r and its successor replica in the chain. Any node that receives such a message, adds r to its `marked` set. This makes that any *accept* message for a subsequent instance is never sent to r , hence r is not able to increment the counter for *accept ack* within those messages.

This happens, even if nodes have not yet locally decided the outcome of instance n_i . This could be problematic if a new leader is meanwhile elected before the outcome of this instance is locally decided (and executed) by every node, since that node could be continuously skipped despite the fact that he was never removed from the system. However, if a leader change happens, the contents of the `marked` set of replicas are removed. This is performed either when the node replies to the *prepare* of the new leader, or when it receives an *accept* message from the new leader (which can be identified locally since the *accept* message will carry a prepare number higher than the last prepare number observed by that replica). This ensures that r receives subsequent *accept* messages for the re-executions of instances $n_j > n_i$ until the leader proposes the `removeNode(r)` in some instance (assuming r remains suspected).

Finally, in the case of concurrent addition or removal of replicas to the system, we note that ChainPaxos executes each addition or removal as an independent operation. We note that the fault detection mechanism may lead replicas to incorrectly suspect other replicas (e.g. due to temporary network failures). In this case, if a replica is incorrectly removed, it can ask to rejoin the system (note that ChainPaxos tries to propagate the remove operation to the node to be removed, as it is still part of the system in that instance). Finally, we note that it is easy to minimize scenarios where replicas ask for the removal of correct replicas by having the leader avoid to either remove replicas that he perceives as active, or removing replicas that were suspected by replicas being currently removed.

B Artifact Appendix

Abstract

The artifact includes the implementation of ChainPaxos, along with the other consensus algorithms that are studied in the evaluation, with instructions on how to launch and test them. These implementations include a simple replicated key-value store that was used to benchmark the algorithms. Additionally, we include our implementation of ChainPaxos in ZooKeeper, which replaces Zab.

For reproducibility, our artifact includes the client-side code that was used in the paper to measure the various performance metrics of the algorithms, along with instructions on how to run it and how to parse and interpret its results.

Scope

The artifact allows executing our consensus algorithm, ChainPaxos, which supports different read execution techniques. Our prototype fully supports the operations related with the integrated membership. The artifact includes everything that was used in the paper: source-code of all solutions; client source-code; scripts to execute the experiments; scripts to generate the plots from the experiment logs and; instructions on how to reproduce all plots.

Contents

The artifact is divided in four parts, which are distributed across four repositories:

ChainPaxos: This repository contains the code for our full implementation of ChainPaxos. Additionally, it includes the key-value store application and the different consensus algorithms that we used to compare against ChainPaxos (in the Figures 4 to 7, 9 and 10). The repository also includes information on how to compile and deploy ChainPaxos.

The source-code in the repository is divided in multiple Java packages, with the following structure: the package *chainpaxos* contains our implementation of ChainPaxos; the code for the key-value store application is on package *app*; packages *frontend* and *common* contain some generic interfaces and classes to uniformize the interaction between the application and all consensus algorithms and; all other packages are named after the consensus algorithms that we used to compare against our solution in the paper.

ZooKeeper with ChainPaxos: This part contains our modified version of ZooKeeper that replaces Zab by ChainPaxos, that was used for the results of Figure 8. The majority of code modifications are contained in the package *chain* that is on the *zookeeper-server* module.

Client-side benchmark: This part contains all the client code that was used in the entire experimental evaluation, both to benchmark the various algorithms using the key-value store, and to benchmark the original ZooKeeper against our version with ChainPaxos. The source-code itself consists of YCSB drivers, one for the key-value store and another for ZooKeeper. The repository also includes the scripts used to perform our experiments, and instructions on how to use them in order to reproduce the results in the paper.

Results parser: Finally, our artifact includes a series of Python scripts that were used to parse the results of each experiment and generate the plots presented in this paper. The *client-side benchmark* repository contains instructions on how to use these scripts.

Hosting

The artifacts can be found in the following locations:

- ChainPaxos
 - <https://github.com/pfouto/chain>
 - master branch
 - commit [72cebf2](#)
- ZooKeeper with ChainPaxos
 - <https://github.com/pfouto/chain-zoo>
 - master branch
 - commit [65a9690](#)
- Client-side benchmark
 - <https://github.com/pfouto/chain-client>
 - master branch
 - commit [ed28200](#)
- Results parser
 - <https://github.com/pfouto/chain-results>
 - master branch
 - commit [e716e4a](#)

Requirements

While the artifact does not have special hardware requirements, all experiments were conducted in the [Grid5000](#) testbed, using the [Gros](#) cluster. The client-side benchmark repository includes instructions on how to reproduce the experiments on this cluster. Furthermore, the same repository provides instructions on how to deploy and run the experiments in any other cluster platform (e.g. on a cloud infrastructure), which requires some additional setup, but should still allow to reproduce all the results in the paper.



CBMM: Financial Advice for Kernel Memory Managers

Mark Mansi

markm@cs.wisc.edu

University of Wisconsin - Madison

Bijan Tabatabai

btabatabai@wisc.edu

University of Wisconsin - Madison

Michael M. Swift

swift@cs.wisc.edu

University of Wisconsin - Madison

Abstract

First-party datacenter workloads present new challenges to kernel memory management (MM), which allocates and maps memory and must balance competing performance concerns in an increasingly complex environment. In a datacenter, performance must be both good *and* consistent to satisfy service-level objectives. Unfortunately, current MM designs often exhibit inconsistent, opaque behavior that is difficult to reproduce, decipher, or fix, stemming from (1) a lack of high-quality information for policymaking, (2) the cost-unawareness of many current MM policies, and (3) opaque and distributed policy implementations that are hard to debug. For example, the Linux huge page implementation is distributed across 8 files and can lead to page fault latencies in the 100s of ms.

In search of a MM design that has consistent behavior, we designed Cost-Benefit MM (CBMM), which uses empirically based cost-benefit models and pre-aggregated profiling information to make MM policy decisions. In CBMM, policy decisions follow the guiding principle that *userspace benefits must outweigh userspace costs*. This approach balances the performance benefits obtained by a kernel policy against the cost of applying it. CBMM has competitive performance with Linux and HawkEye, a recent research system, for all the workloads we ran, and in the presence of fragmentation, CBMM is 35% faster than Linux on average. Meanwhile, CBMM nearly always has better tail latency than Linux or HawkEye, particularly on fragmented systems. It reduces the cost of the most expensive soft page faults by 2-3 orders of magnitude for most of our workloads, and reduces the frequency of 10-1000 μ s-long faults by around 2 orders of magnitude for multiple workloads.

1 Introduction

Datacenter workloads present new challenges to kernel memory management (MM). MM encompasses a large collection of kernel mechanisms and policies to allocate and map physical memory. Cumulatively, they comprise a complex set of

tradeoffs that, when poorly navigated, lead to poor performance or unexpected behavior. For example, we found that for some workloads on Linux, a soft page fault lasting 25ms occurs every 100ms. This drastic tail latency is due to memory compaction or reclamation when attempting to allocate a huge page – a misnavigated tradeoff. Many applications would violate response latency objectives if one request per 100ms takes 25ms due to a page fault. As a result, Redis, MongoDB, and others advise users to disable Linux’s Transparent Huge Page (THP) feature [2, 3, 4, 7, 50]. Table 1 lists other examples of MM policies and their potential pathologies.

The hardware and software in modern datacenters differ vastly from those in use when MM techniques were first designed. Increased memory capacities allow more workloads to run but bring challenges too: huge page management becomes more critical due to increased reliance on TLB performance, but memory fragmentation and huge page management overheads also increase with memory capacity [36]. Datacenters also prioritize *tail* latency as a key service-level metric, in addition to median latency and throughput [19]. Datacenter behavior must be consistent, i.e., low variance, without compromising performance metrics to satisfy service-level objectives and efficiency goals.

Unfortunately, current MM designs often fall short of modern computing needs by exhibiting inconsistent, opaque behavior that is difficult to reproduce, decipher, or fix. These issues come from three key limitations.

First, kernel MM must predict workload behavior in an information-poor environment. Current MM designs rely on online measurements, particularly page table access/dirty bits and the frequency and location of page faults. Unfortunately, this information is expensive to collect and low bandwidth. For example, Google uses access bits to detect idle memory [15], but other work finds them insufficient to predict TLB miss overheads accurately [38], even though they can cost up to 11% of CPU cycles to collect [15]. Other data collection mechanisms induce additional page faults [16, 24]. Recent work uses performance counters in kernelspace [38], but currently available counters are hardware-thread-oriented

Policy	Goal	Pathology
Huge Page Allocation	Reduce TLB misses and page faults	Bloat memory usage if not all memory is used; increase page fault latency if compaction is required
Eager Paging [29]	Move page fault latency to allocation time, saving time later	Bloat memory usage if not all memory is used
Background Compaction	Reduce memory fragmentation and huge page fault latency	Increase CPU overhead
Background Zeroing	Reduce page fault latency	Increase CPU overhead
Idle Page Reclamation [32, 47]	Improve memory utilization	Increase overhead to fault reclaimed pages back in; increase CPU overhead to choose pages to reclaim

Table 1: Different MM policies and their goals and pathologies

and do not provide the detailed spatial information useful for most MM policies.

Second, current MM designs often ignore the cost of various MM operations, leading to inappropriate policy decisions. For example, Linux allocates a huge page when a memory region is first touched; however, we find that allocating and zeroing a huge page costs 10^6 cycles in the best case. Thus, promoting a page that averts $\leq 10^6$ cycles worth of TLB misses and page faults actually *regresses* performance, but the kernel does not account for this cost.

Third, current MM designs are implemented as disjointly acting policies distributed throughout the kernel that are hard to debug. For example, code implementing Linux’s huge page policies is scattered across more than eight files (and numerous functions), mixed with unrelated code. Users and developers observe erratic slowdowns without indication of what causes them or how to address them. They often resort to suboptimal coarse-grain solutions, such as disabling huge pages [2, 3, 4, 7, 50]. By distributing and obscuring policy-implementing code, current kernel MM implementations make it difficult for both kernel and userspace developers to decipher system behavior. This opaque system implementation and its consequent opaque behavior is a primary obstacle to improving kernel MM performance, consistency, and debuggability.

In search of a MM design that has consistent behavior, we designed Cost-Benefit MM (CBMM). CBMM reflects that all kernel MM operations have a cost and a benefit to userspace, and it estimates them using empirically based cost-benefit models to guide MM policy decisions. By explicitly modeling cost and benefit, CBMM is more cost-aware than current designs, so it makes fewer pathologically bad policy choices. Also, CBMM augments online statistics with offline-aggregated profiles to improve the quality of information available to the kernel. CBMM simplifies policy debugging and enables incremental performance improvement by centralizing models in a new kernel component: the *estimator*. To understand and fix anomalies, one must only understand the model inputs to determine the cause of a policy decision.

Our prototype implements models for huge page promotion, asynchronous page prezeroing, and eager paging [29], based on an in-depth analysis of huge page behavior and soft page faults. At runtime, they may make use of in-built em-

pirically based assumptions (e.g., about average TLB miss latency), online information (e.g., the current number of free pages), or offline-aggregated profile information (e.g., fine-grained information about huge page benefits). We focus on *first-party* datacenter workloads – software run by service providers in their own datacenters – as they are highly controlled and relatively stable over time, allowing better profiling and modeling [1, 6, 10, 12, 27, 42, 44, 45].

CBMM improves system consistency; it nearly always has better tail latency than Linux or HawkEye, particularly on fragmented systems. It reduces the cost of the most expensive soft page faults by 2-3 orders of magnitude for most of our workloads, and reduces the frequency of 10-1000 μ s-long faults by around 2 orders of magnitude for multiple workloads. Meanwhile, it has competitive performance with Linux and HawkEye, a recent research system [38], for all the workloads we ran, and in the presence of fragmentation, CBMM is up to 35% faster than Linux on average – all while using no more huge pages than Linux or HawkEye in most cases.

2 Motivation: Evaluating Current Behavior

To quantify the extent of these challenges and inform our design, we do an in-depth analysis of two important kernel MM code paths, huge page management and page fault handling. Our experimental setup is described in Section 5.1. Full results are available [in our artifact](#).

2.1 Measuring Huge Page Benefits

Huge pages speed up many workloads, but nobody has quantified the impact of workload behavior on the amount of speedup it receives from huge pages. Thus, we measure the fine-grained benefit of huge pages as described in Section 4.1. To avoid invasive instrumentation and a detailed survey of workload implementations, we measure huge page benefits from the perspective of the kernel: for each workload, we divide the address space into 100 equally sized ranges, excluding unmapped regions, and repeatedly run the workload backing one range at a time with huge pages.

Figure 1 shows the results. Each point on the x-axis represents one range, such that the x-axis represents the virtual

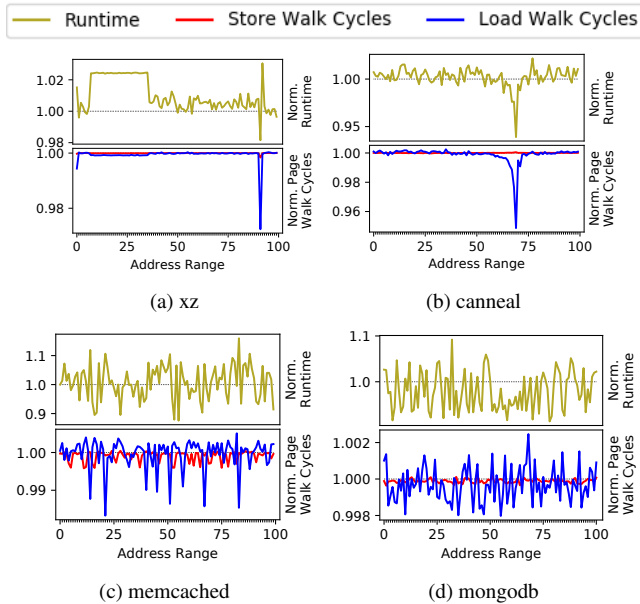


Figure 1: Runtime and usermode cycles spent in page walks for each address range, normalized to no huge pages (lower is better). Note the varying y-axes.

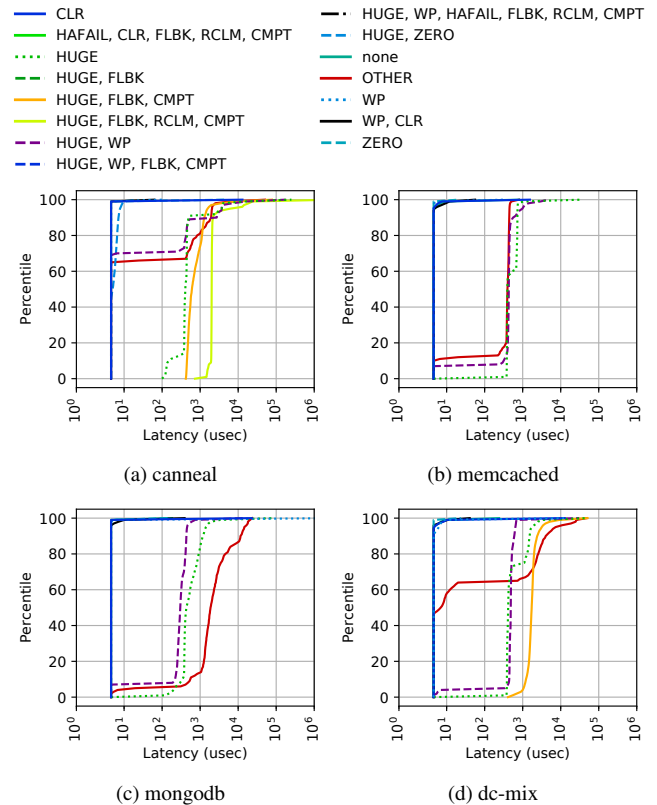
address space. The top y-axis shows the normalized performance compared to no huge pages. The bottom y-axis shows the normalized percentage of time spent in usermode page walks (i.e., TLB misses) for loads and stores.

The impact of huge pages varies extensively between workloads. *xz* and *canneal* primarily see improvements in *load* page walk cycles from backing particular regions of the address space corresponding to hot data structures. *memcached* and *mongodb* produce noisy results because of randomness in the workload. The magnitude of impact ranges from about 0.25% in *mongodb* to almost 7% in *canneal*.

Another benefit of huge page usage is fewer page faults. We found that they have only a minor contribution to performance (e.g., less than 1.2% of execution time for *canneal*).

Also, the relationship between runtime improvement and reduction in page walk cycles is not straightforward. For all workloads, runtime improvement is loosely correlated with either load or store page walk cycles. Strong effects on either load or store page walk cycles tended to be reflected in runtime, as seen in *xz* and *canneal*, but the magnitude of that effect varies. Small changes in page walk cycles often have no apparent effect on runtime.

Discussion Huge page impact varies greatly by workload, including the type and location of impacted memory accesses and the magnitude of impact. Additionally, the relationship between page walk cycles and runtime is complex, illustrating the challenge of huge page management given the limited, low-quality information available to the kernel at runtime such as CPU performance counters and page referenced bits. For example, *dc-mix* (not depicted) benefits from backing



Flag	Description
CLR	Memory was zeroed (usually during allocation).
CMPT	Allocator used memory compaction.
FLBK	Allocator used a fallback path during the page fault.
HAFAIL	Attempted to allocate a huge page and failed.
HUGE	A huge page was mapped.
PREZ	Allocator allocated a prezeroed page (CBMM only).
RCLM	Allocator used direct reclamation.
WP	Page fault due to write to a write-protected page.
ZERO	A (shared) zero page was mapped.

(e) Subset of bitflags for page fault tracing.

Figure 2: CDF of Linux soft page fault latency by type of page fault. Not all page fault types occur in all workloads.

individual regions with huge pages, but when THP is turned on, it sees a net regression in performance due to the overhead of compaction. CBMM aims to mitigate this problem by supplying the kernel with higher-quality information.

2.2 Soft Page Fault Latency Breakdown

We instrument Linux’s page fault handler to trace sources of page fault latency. Page fault tracing allows us to characterize system-wide costs, such as the cost of zeroing memory. We identified a set of events that occur during page faults and associate each with a bitflag (Figure 2e). Our instrumentation records the total time of the page fault, the time to allocate memory, and the time to clear/copy memory contents.

We record the flags and timing of all events longer than 10^4

cycles, and a count of shorter events, allowing us to compute the proportion of all page faults with each set of flags. We exclude hard page faults from our results, as they incorporate other kernel subsystems (e.g., block I/O, file systems). Our tracing records the total time to handle a page fault, but on x86 the handler can be interrupted in favor of another task, which inflates the latency of the page fault. This is rare in most workloads except `mongodb`, which uses a userspace asynchronous I/O framework and thread pool; even though a page fault handler may be descheduled for a while, userspace requests continue to make progress because of userspace threading.

Figure 2 shows the soft page fault latency breakdown for multiple workloads. For each distinct set of flags, the CDF of page faults with those flags is plotted. Note that the x-axis uses a log scale. The plot includes samples lower than the threshold by treating them as if they all took 10^4 cycles (in reality, most are faster than that). The figure shows results on a freshly booted, unfragmented system, which represents best-case performance; we also recorded results on fragmented system, and found them significantly worse for all workloads.

The results indicate three challenges current MM designs face. First, applications trigger a wide variety of kernel behaviors. Each of the 15 flag-sets of Figure 2 is a different combination of code paths. Second, different paths have very different latencies but are relatively consistent across workloads. For example, even in this best case, a huge page consistently takes hundreds of microseconds to be allocated (`HUGE` in the figure) due to zeroing overhead. Third, many pathological code paths execute that do not benefit applications. Most notably, a huge page allocation may invoke a fallback path (`FLBK`), which transitively invokes compaction (`CMPT`) or reclamation (`RCLM`). Worse, the fallback may fail (`HAFAIL`), resulting in a base page allocation after all. In `canneal` (Figure 2a) and `dc-mix` (Figure 2d), these fallback paths can take *dozens or hundreds of milliseconds*. In contrast, an Amazon search for “DRAM” completes in only 900ms from our office.

Discussion Linux’s fallback algorithms are severely cost-unaware and make system behavior inconsistent: invoking compaction or reclamation almost certainly outweighs any benefits of using a huge page. Also, the high cost of zeroing suggests that memory prezeroing (Section 4.2) may be a useful optimization to make huge pages more useful. Currently, if an average TLB miss costs around 30 cycles, then a huge page must avert over 33,000 TLB misses to pay for itself. These results highlight the need for cost-aware MM policies.

3 Cost-Benefit Memory Management

We created the Cost-Benefit Memory Manager (CBMM), which has several goals:

- Improve kernel MM behavioral consistency,
- Match existing systems’ performance,
- Improve the debuggability of policy decisions,

- Allow incremental improvement of individual policies.

Our key insight is that all MM decisions incur a cost against and provide a benefit to userspace. For example, huge page promotion averts TLB misses but may require zeroing or compacting memory. In CBMM, policy decisions follow the guiding principle that *userspace benefits must outweigh userspace costs*. By applying this principle uniformly, CBMM significantly improves consistency over Linux and HawkEye [38], while matching their performance. We design models for three important kernel MM policies: huge page promotion, asynchronous page prezeroing, and eager paging [29].

CBMM introduces a new component, the *estimator*, to the kernel. It estimates the cost and benefit of a given MM operation whenever a policy decision is needed. If $cost < benefit$, the kernel decides to execute the operation.

The estimator makes estimates based on empirically derived cost and benefit models. Models can optionally use live metrics and/or pre-aggregated profiling information. Such pre-aggregated information can mitigate the lack of high-quality online information. Meanwhile, CBMM explicitly estimates MM operation costs, improving cost-awareness.

In current MM implementations, policy decisions are scattered across the kernel, making it difficult to coordinate their actions and difficult to debug anomalous behavior. In contrast, CBMM invokes the estimator at decision points, which predicts the cost and benefit of taking an action. This centralizes decision making and explicitly marks policy decisions points. It also makes coordination between policies easier.

A key requirement of CBMM is that the system behavior can be modeled and/or profiled. This requirement holds for many first-party datacenter workloads, which often run with high redundancy for long amounts of time [1, 6, 10, 12, 27, 42, 44, 45], giving ample opportunity to observe and instrument a workload before applying policies to them.

3.1 The Estimator

In CBMM, the MM subsystem invokes the estimator at places in the code where policy decisions need to be made. We call these places in the code *decision points*. It uses models to estimate the cost and benefit of a particular MM operation and returns the estimates to the decision point, which executes the operation if $cost < benefit$.

When a decision point invokes the estimator, it passes information to the estimator about the type and parameters of the operations. For example, the decision point would pass the address to consider promoting or a number of pages to attempt to prezero. The estimator acts as a black box that returns a cost and benefit estimate for the given MM operation and parameters. In CBMM, costs and benefits are computed in units of time saved or lost by userspace, which usually corresponds closely to user objectives. In particular, CBMM uses the *rate* of time saving/loss over some horizon, as many datacenter workloads run continuously.

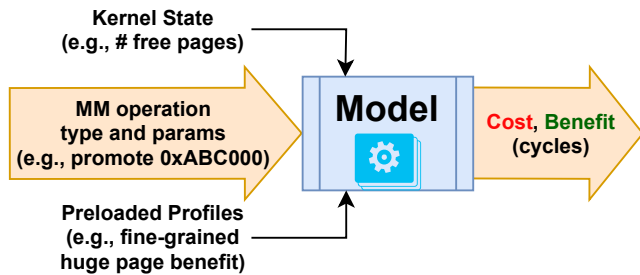


Figure 3: CBMM model inputs and outputs.

3.2 Cost and Benefit Models

Internally, the estimator comprises a collection of cost models and benefit models for different MM operations. Each model is built out of simpler submodels that estimate one cost and/or benefit well; the submodel results are added to produce the overall result. This allows reuse of submodels for different decision points, simplifying implementation and leading to more consistent behavior across decision points. For example, our huge page cost-benefit models were useful in both the page fault handler and `khugepaged`, the background promotion daemon, and our model for estimating the cost of running a daemon could be used for multiple daemons in the future.

Concretely, models manifest as C code in the estimator (in the kernel); in Listings 1 - 3 (discussed further in Section 4), we show the models in our prototype of CBMM. Each (sub)model is a self-contained black box that takes information from the decision point, combines it with information from the ambient kernel state and *preloaded profiles* – files loaded into the kernel that supply information about application-specific behavior – and outputs an estimate, as shown in Figure 3. The additional input from the kernel state and preloaded profiles allows the models to be more context-aware and to make use of higher-quality information about workload behavior.

Performance Debugging Unlike current heuristics, CBMM isolates policies to specific cost and benefit models; their inputs and outputs can be observed, and they can be improved in a single place, easing performance debugging in CBMM compared to Linux. A central idea behind CBMM’s debuggability is the ability to observe and control the inputs to models. Thus, while models can make use of any kernel or hardware state, they should use only state that has an intuitive interpretation, rather than internal implementation metrics. For example, our huge page promotion model takes into account whether any prezeroed huge pages are available and uses a profile to determine the worth of promoting a page. In contrast, internal implementation metrics give limited information about the origin of their values and how to cause them to change, making bug fixing difficult; for example, Linux’s page reclamation algorithm uses an obscure combination of page table bits, bit flags in the `struct page`, and what list a page happens to be on [9].

Model Development in CBMM can be done iteratively by beginning with a simple model and refining it as needed. For example, Listing 2 shows our asynchronous prezeroing model. Initially, we only accounted for the zeroing time of the daemon, but we found that this led to high lock contention on the allocator, so we refined the model to account for contention.

In designing our models, we found that benefits tend to be application-specific, whereas costs tend to be system-specific. For example, each application tends to benefit differently from huge pages, but the cost to allocate a huge page is application-independent and depends more on the state of the system allocator. As a result, our benefit models tend to use preloaded profiles, whereas our cost models tend to query kernel state.

Models necessarily make assumptions to simplify implementation and to make their execution cheaper than the actual MM operations. We based our assumptions on our empirical measurements, unlike many existing heuristics, which rely on intuitive simplicity or common-case optimization. For example, unlike Linux, CBMM does not blindly assume huge pages improve performance; rather, it incorporates the cost of promotion as measured by our experimental analysis and uses empirically derived profiles to estimate the benefit of promoting a particular memory region. Notably, CBMM improves system behavior even with imprecise profiles, as we will show in Section 5.5, making it practical to start with a simple model and refine it over time.

3.3 Preloaded Profiles

Different applications respond differently to MM policies, and kernels currently lack high-quality information with which to predict workload behavior. Preloaded profiles are files loaded into the kernel when starting a process (e.g., by a cluster manager) to provide models with information about a process’s behavior. They allow the estimator to benefit from offline processing for particular policy decisions. In contrast, prior methodologies resort to measuring inaccurate and expensive proxy statistics such as page fault counts or page access bits.

In CBMM, preloaded profiles specifically provide *spatial, per-process information*; that is, they provide information about regions of a single address space at arbitrary granularity as small as a 4KB page. For example, a profile may specify per-region reduction in page walk cycles from use of huge pages, or a bit indicating whether a page is likely to be touched or not. Models can query this information when making cost and benefit estimates. For example, to estimate the benefit of using a huge page, a model may incorporate the number of averted page walk cycles, or to determine whether to eagerly allocate memory or use copy-on-write, a model may incorporate information about the likelihood of memory accesses. This structure for preloaded profiles, while simple, is quite useful because many MM policies make *spatial* decisions, such as whether/how to map/unmap/remap a memory region. However, CBMM’s design is flexible enough to admit

future extensions to profiles. For example, it may be desirable to account for phases of workload activity or to apply profiles at different granularities, such as per-thread or system-wide.

Profile Management. While CBMM still has benefits even when profiles are imprecise (see Section 5.5), changes to data structures or algorithms could result in changing memory reference patterns. Thus, a natural future extension of CBMM is automating profile generation and management.

First-party datacenter workloads often run continuously and redundantly, so profiling could be automated and centralized at the cluster level. Recent work from industry suggests a trend of large-scale profiling and centralized planning [32, 35, 42] and demonstrates the feasibility of such an approach. We have done preliminary exploration and believe that the huge page methodology of Section 4.1 can be run in a distributed, automated manner by cluster managers.

3.4 System Management

We implemented models directly in the kernel source, but in principle, they could be implemented via another mechanism, such as kernel modules. Our models are application-agnostic (but can be customized if needed, like existing code), so application developers do not need kernel access. Many service providers have kernel teams that could maintain this code. Meanwhile, profiles are application-specific, and application developers can use existing configuration/deployment systems to schedule/manage/store/secure/deploy profiles without special privileges. The kernel memory overhead of profiles depends on profile resolution/detailedness. In Section 5, our largest profile is $\sim 170\text{KB}/\text{process}$ and most profiles are $< 50\text{KB}/\text{process}$.

Our implementation uses `procf`s files to load profiles, but any user-kernel communication mechanism could be used. Also, in principle one could load models through boot time configuration, similar to Facebook's SoftSKU system [42].

3.5 Discussion

CBMM addresses the (1) information-poverty, (2) cost-obliviousness, and (3) disjointed implementation of existing MM policy implementations, while other alternatives only partially address them. For example, interfaces such as `madvise` are coarse-grained, whereas workload memory access patterns can vary significantly within a region, as shown in Section 2.1. Merely disabling overly-aggressive policies, such as Linux's THP or defragmentation policies, harms workloads that require many huge pages, as we will see in Section 5. Additionally, it is difficult to modify existing policies to target different performance goals because their implementation is often distributed across many files, such as Linux's huge page policies. CBMM mitigates all three challenges by making costs and benefits explicit and centralizing policy decisions. Finally, more research is also needed to determine how

far CBMM's design can be generalized to other areas of the kernel, such as scheduling, filesystems, or I/O management.

4 Implementation

We implement CBMM based on Linux 5.5.8 for three kernel MM policies: huge page management, asynchronous prezeroing, and eager paging [29]. We implement the estimator and its models, along with related debugging interfaces, code for parsing profiles, and other infrastructure in 1159 lines of C in the kernel in a new and self-contained file. Additionally, we add 87 lines of instrumentation throughout the page fault handler and page allocator for page fault tracing (see Section 2.2). We add 10 calls to the estimator throughout the MM subsystem; each is self-contained and consists of about 10 lines of code to initialize a struct, make a function call, and respond to estimates. Asynchronous prezeroing is implemented in a kernel module from Hawk-Eye. We modify the module to run in a kernel thread and to query the estimator before running. Our version of the module is 196 lines of C. Our implementation is available at <https://github.com/multifacet/cbmm-artifact>.

4.1 Huge Page Management

Background Huge page support in current kernels can be either manual and automatic. A primary challenge is choosing memory regions to promote: the kernel must determine which memory regions would see enough benefit from huge pages.

Manual management allows applications to directly request huge pages for certain memory regions, but it requires modifying the applications, which is often impractical (e.g., Java does not expose a way to easily control memory allocation). Moreover, modern datacenter workloads are multi-programmed and diverse in behavior, requiring centralized coordination during resource allocation [31]. In contrast, automatic huge page management is a kernel feature that promotes application memory transparently to applications. This allows unmodified applications to benefit from huge pages but cannot make use of application-specific domain knowledge.

CBMM combines both the generality of automatic management and the application-specific knowledge of manual management. In contrast, current kernels either have only a manual system (e.g., Windows) or use simplistic heuristics to power an automatic system. For example, Linux's THP aggressively tries to promote on the first page fault to the huge page, potentially leading to memory bloat and increased page fault latency. FreeBSD waits for a specific percentage of the huge page to be touched before promoting. Various research systems use a mix of page access bits, performance counters, LRU lists, and trial-and-error [31, 38, 49] with mixed success.

Model Listing 1 shows CBMM's model for huge page promotion. It is used in both the page fault handler and `khugepaged` to decide whether to promote a page. We built

this model based on our analysis of huge page promotion overheads. It makes a number of simplifying assumptions when estimating both the cost and benefit, most notably that the cost is dominated by the allocation and zeroing time and that compaction and reclamation have a large fixed cost. We choose to ignore other costs in our model, such as caching, mapping changes, or potential memory bloat, but CBMM allows models to be iteratively improved over time.

```

void hpage_promo_model(u64 addr, mm_cost_delta *cost)
{
    // COST. Simplify using assumptions.
    // - Alloc is free if have free zeroed pages.
    // - Alloc cost is zeroing if have free unzeroed.
    // - Alloc cost is 2^32 if need to free mem.
    // - We don't care what node it is on.
    // - Constant prep costs (zeroing or copying), ~100us

    // `have_free_hpage` checks the allocator free list.
    const u64 EXPENSIVE = 1ul << 32; // cycles
    enum free_hpage_status fhps = have_free_hpage();
    u64 alloc_cost = fhps > fhps_none ? 0 : EXPENSIVE;
    u64 prep_cost = fhps > fhps_free ? 0 : 100*FREQ_MHZ;
    cost->cost = alloc_cost + prep_cost;

    // BENEFIT = averted TLB miss cycles from profile.
    cost->benefit = compute_hpage_benefit(addr);
}

```

Listing 1: CBMM huge page cost-benefit model.

Profiling Our methodology generates for each workload a mapping from virtual memory regions (i.e., ranges of virtual addresses) to the number of averted usermode page walk cycles when the region is backed by huge pages. We modify the Linux kernel to give precise control over promotions. We then repeatedly run a given workload varying the set of promoted pages. We additionally run the workload with no huge pages as a baseline. We use hardware performance counters to measure the number of TLB misses, the number of cycles spent in pages walks, and the overall cycle count for kernelspace and userspace execution. We then take the difference in overhead and overall runtime between any given set of pages and the baseline. The size of the sets of promoted pages can be varied to tradeoff profiling time with precision. Our prototype uses the offset into allocation zones instead of virtual addresses, so that profiles tolerate Address Space Layout Randomization.

Broadly, we found that our workloads could be categorized as *high-processing* or *low-processing*. *High-processing* workloads, such as `xz`, `canneal`, or `mcf`, heavily process their input data to produce internal data structures; their memory access patterns are driven by computation over these data structures. *Low-processing* workloads, such as `memcached`, `mongodb`, and `dc-mix` (see Section 5.1), often do little more than data storage and retrieval, so their access patterns are driven by client request patterns. We found that we can reliably distinguish between high- and low-processing workloads using the skewness¹ of the distribution of averted page walk cycles. High-processing workloads often have a small number of highly-impactful memory regions, so they have a high positive skew ($skew > 2$ seems to work empirically). When generating

¹skewness is a statistical measure of distribution asymmetry.

profiles for low-processing workloads, we assigned all regions a benefit equal to the mean benefit measured empirically. For high-processing workloads, we assigned each region the benefit it individually demonstrated.

At runtime, we can supply a profile to the kernel in the form of a CSV file that lists virtual address ranges and their benefit from huge pages. Our implementation aims to demonstrate the potential of our approach while remaining simple to implement. We do not attempt to account for phases in workload behavior, but our design is amenable to such an upgrade in the future by repeating the profiling process at multiple points during the workload’s execution. We assume the workload size is stable but can handle other input changes; in Section 5, we use randomized inputs for most workloads.

4.2 Asynchronous Prezeroing

Background We examine *asynchronous prezeroing* as a means of improving the latency of large physical memory allocations. Asynchronous prezeroing clears free pages using a background daemon to save time during a page fault when it would slow down userspace programs. Our analysis indicates that prezeroing would reduce the cost of a huge page by almost two orders of magnitude.

Prezeroing has fallen out of favor because the primary cost of zeroing 4KB pages is cache misses, but prezeroing pages leaves them cold when users access them, so latency is merely shifted to userspace [8, 46]. Recently, Panwar et al. find prezeroing is beneficial for huge pages and use non-temporal store instructions to avoid cache pollution [38]. However, their approach requires hand-tuning to avoid excessive CPU usage or lock contention on the page allocator. CBMM adapts their prezeroing implementation with a model to determine when and how long to run, eliminating the need for hand-tuning.

Model Listing 2 shows CBMM’s model for running the asynchronous prezeroing daemon. The model makes numerous assumptions; most importantly, it assumes that CPU time is free unless taken away from userspace (i.e., the system is not idle) and that the chief costs of prezeroing are the execution time of zeroing and contention on the allocator lock, rather than cache pollution. This matches our own analysis and observations while working on CBMM. The chief benefit of prezeroing is to move zeroing overhead out of the critical path of huge page promotion. For simplicity, we assume a constant processor frequency over short time windows, even though the frequency varies.

Also, this model exemplifies CBMM’s iterative approach to building models. We started with a model that only accounted for CPU time and potential huge page allocations. As we ran experiments, we discovered the lock contention and improved the model to account for it by adding the lines labeled as `COST` of lock contention in Listing 2, resolving the performance issue. The entire process took less than a day of debugging, measurement, and implementation.

```

void prezeroing_model(mm_action *action,
                    mm_cost_delta *cost)
{
    // COST of the runtime itself... Assume:
    // - Don't care about NUMA nodes.
    // - Zeroing costs ~10^6 cycles.
    __kernel_ulong_t cpu_load = get_avenrun();
    int ncpus = num_online_cpus();
    const u64 HPAGE_ZERO_COST = 1000000;

    // ncpus > cpu load average => idle cpu, free to run.
    if (ncpus > cpu_load) {
        cost->cost = 0;
    } else {
        cost->cost = HPAGE_ZERO_COST * action->prezero_n;
    }

    // COST of lock contention. Assume:
    // - Cost of lock acquisition = ~150cyc, do it 2x.
    // - Lock is unheld for ~1ms/horizon => free locking
    const u64 UNHELD = FREQ_MHZ * 1000; // cycles
    const u64 SINGLE_CS = 150; // cycles
    const u64 crit_sect_cost = SINGLE_CS * 2; // cycles
    const u64 nfree = UNHELD / crit_sect_cost;
    cost->cost += (action->prezero_n > nfree
        ? action->prezero_n - nfree : 0) *
        critical_section_cost;

    // BENEFIT. Assume past usage predicts future usage.
    u64 recent_used = mm_estimated_prezeroed_used();
    cost->benefit = min(action->prezero_n, recent_used)
        * HPAGE_ZERO_COST;
}

```

Listing 2: CBMM async prezeroing cost-benefit model

4.3 Eager paging

Background Eager paging allocates physical memory upon user request, rather than lazily on a page fault (the default) [29]. It enables large contiguous physical memory allocations, which are easier to back with huge pages and enable useful hardware optimizations [29, 37, 40, 43, 49]. However, a drawback to eager paging is memory bloat if the workload does not use all the allocated memory [29]. Preloaded profiles unlock this optimization while avoiding memory bloat.

Model Listing 3 shows CBMM’s model for eager paging, which is invoked by `mmap` or `brk` system calls. It uses a preloaded profile to determine which subregions will be touched and assumes that the model has perfect knowledge, allowing it to ignore the cost of potential bloat. If more than one page is being eagerly allocated, we create opportunity for contiguous allocation.

```

void eager_paging_model(vm_area_struct *mmap_region,
                      mm_cost_delta *cost)
{
    // ASSUME: past usage predicts future; use profile.
    // COST: time to create new page.
    const u64 PF_NEW_PAGE = FREQ_MHZ * 10; // cycles
    struct range *ranges = prev_touched(mmap_region);
    cost->cost = len(ranges) * PF_NEW_PAGE;

    // BENEFIT: time to create new page, coalesced faults
    const u64 PF_CS = 300; // cycles
    cost->benefit = len(ranges) * PF_NEW_PAGE
        + (len(ranges) - 1) * PF_CS;
}

```

Listing 3: CBMM eager paging cost-benefit model

Profiling We profile eager paging behavior by periodically reading the `/proc/<pid>/pagemap` file while the workload is

running. This file contains information about memory mappings for the given process and allows us to detect which virtual memory regions have been faulted in. Pages that were faulted in during the execution are noted in the profile, and the model assumes they will be faulted in again in the future.

5 Evaluation

CBMM seeks to improve consistency while matching or exceeding the performance and efficiency of existing systems. We evaluate CBMM along multiple axes. First, we evaluate the page fault latency of CBMM to understand its consistency compared to Linux and HawkEye. Second, we measure the end-to-end performance of CBMM. Third, we look at the efficiency of CBMM’s use of huge pages. Finally, we evaluate the generality of our approach by looking at the sensitivity of performance to profile changes.

5.1 Methodology

Table 2 describes our workloads. They represent a variety of software behaviors and exercise the kernel in different ways. `mongodb`, `memcached`, and `dc-mix` are memory-intensive workloads common in datacenters. `mongodb` and `memcached` are data stores, and `mongodb` is I/O heavy and makes use of the page cache. `dc-mix` induces memory pressure and tries to simulate a real system in which a server, device driver, and batch job are using system resources. We drive the data stores in these workloads using YCSB [17] with different read-write ratios to increase the variety of MM behavior. `mcf`, `xz`, and `canneal` are computational workloads. We scale up the inputs of `xz` and `canneal` to use more memory. In all experiments with server applications (e.g., `memcached`, `redis`, `mongodb`), we run the client program on the same machine as the server, so as not to measure network effects. We run each workload with its default number of threads and pin all workloads to one NUMA node to reduce variation caused by NUMA effects. To reduce noise, we run each experiment 5 times and report the median results. For all workloads except `mcf` and `xz`, the input is randomized and changes between executions of the workload. For `xz`, we use the native input to generate a profile and use a custom input when evaluating performance.

All experiments run on CloudLab [41] c220g5 machines with two Intel Xeon Silver 4114 (10C/20T, 2.2 GHz, Skylake 2017), 192GB 2666MHz DDR4 ECC DRAM, and a 480GB SAS SSD. We set the CPU scaling governor to `performance`. *Unless otherwise noted, we do not tune our systems at all*; the results represent CBMM’s “out of the box” behavior.

We replace the system allocator with `jemalloc`, which is better in a datacenter setting and is used by Facebook [23]. All experiments run on CentOS 7.8.2003 with the relevant kernel. We disable Meltdown and Spectre [30, 34] mitigations, which cause severe performance degradation. We use unmodified

Workload	Description	Input	Peak Mem
xz	data compression [5]	profiling: native input, eval: custom scaled up input	150GB
mcf	combinatorial optimization, scheduling [5]	native input	3GB
canneal	simulated annealing, chip routing [13]	custom input, randomly generated each time	150GB
mongodb	KV store	YCSB driver [17], 75%W-25%R	150GB
memcached	in-memory KV store	YCSB driver [17], 1%W-99%R	150GB
dc-mix	redis (KV store), memhog (microbench., creates fragmentation), metis (in-memory map-reduce) [28]	redis: YCSB driver [17], 50%W-50%R; memhog: N/A; metis: built-in	165GB

Table 2: Description of Workloads – their behavior, inputs, and peak memory usage.

Linux 5.5.8 with Transparent Huge Pages enabled as our baseline. We configure CBMM similarly to Linux but we preload a profile of huge page benefits and eager paging, as derived in Sections 4.1 and 4.3. We also compare against HawkEye [38], a state-of-the-art research huge page management system based on Linux 4.3. We configure HawkEye as in its paper, including its prezeroing daemon. We ran experiments against stock Linux 4.3 and found that it performs within 15% of Linux 5.5.8 on average (see Figure 5). To measure page fault latency in HawkEye, which runs on a different kernel without our instrumentation, we use eBPF to instrument the `handle_mm_fault` function, which represents the main portion of the page fault handler. We found that `canneal` crashes with a segfault on HawkEye when the system is unfragmented, so we omit that experiment from results.

Fragmentation We run each workload on a freshly rebooted system and on a preconditioned system. Preconditioning aims to simulate a long-running datacenter environment by inducing external fragmentation, which hinders large physical memory allocations, such as huge pages.

We had difficulty identifying a reproducible fragmentation methodology. We attempted to reuse techniques from prior work [38, 49, 51] and also made several attempts at our own methodologies with little success; on Linux, deferred freeing of physical memory and kernel daemons such as `kcompactd` and `kswapd` cause variable results. Also, each methodology preconditions machines in a different way, none of which is obviously more realistic than the others.

For our evaluation, we choose a simple methodology derived from prior work [18, 49, 51]. We enable `CONFIG_SLAB_FREELIST_RANDOM` and `CONFIG_SHUFFLE_PAGE_ALLOCATOR` when compiling the kernel and add a `sysfs` file that triggers shuffling of the kernel physical memory free lists. To precondition the system, we reboot and then trigger free list shuffling. Then we run a program that allocates all system memory (with `mmap(MAP_POPULATE)`) and frees all but the first page of each 2MB region before sleeping for the duration of the workload. This methodology is simple and yields comparable results to other methodologies. We measure the Free Memory Fragmentation Index [25, 31, 38, 51] after preconditioning but before the workload begins. On CBMM and HawkEye, preconditioning consistently leaves around 183GB of free memory with 99% fragmentation. On Linux, half of runs experience similar results, but in the other half, deferred page freeing causes < 2GB of memory to be considered free,

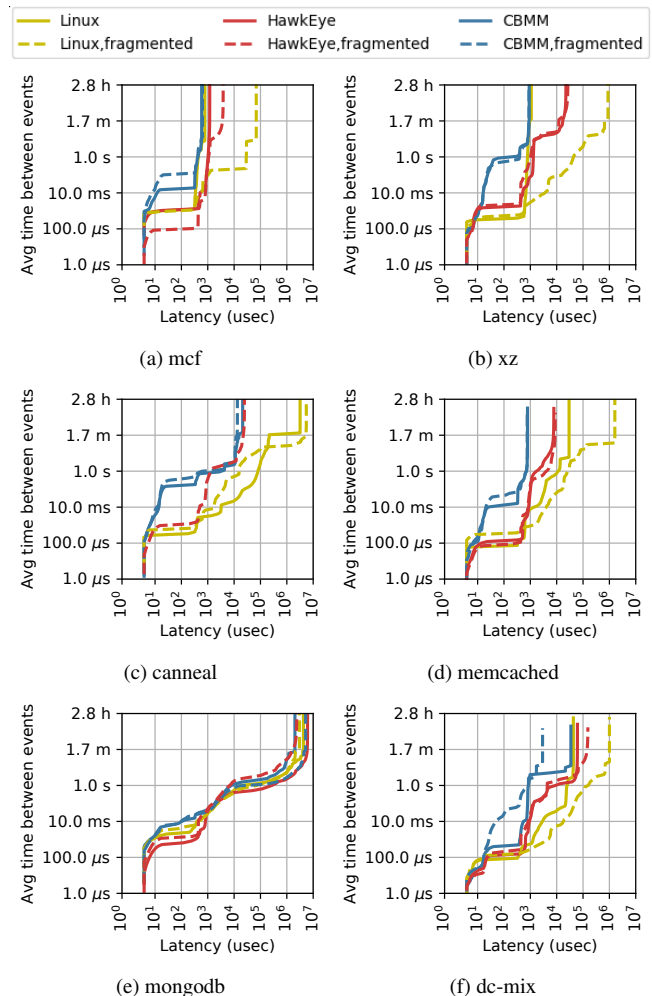


Figure 4: Soft page fault tail latency distribution on each system, weighted by page fault rate. A point (x, y) on the plot indicates that a fault of latency $\geq x$ happens at an interval of $\geq y$ on average.

making it difficult to measure fragmentation.

5.2 System Behavioral Consistency

Figure 4 shows tail latency on each kernel without fragmentation (when latency should be lowest); note the log x- and y-scales. To account for differences in the frequency of page faults due to differing MM decisions, we show the average in-

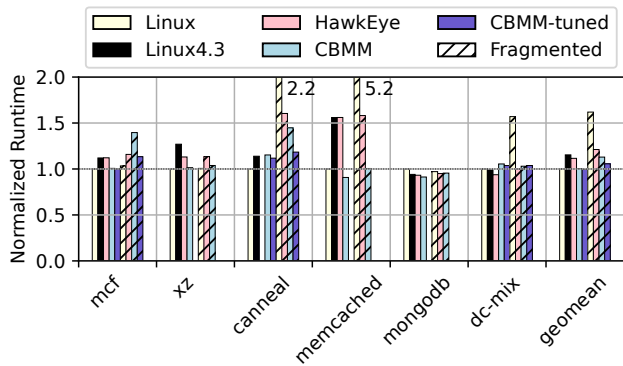


Figure 5: Runtime of workloads on each kernel, normalized to Linux with THP without fragmentation (lower is better).

terval between events, rather than the percentile on the y-axis.

Unlike Linux (Figure 2d), CBMM rarely attempts an expensive fallback path (e.g., compaction or reclamation) during huge page promotion, even under fragmentation; allocation failures usually result in the allocation of base pages. CBMM often experiences more page faults than Linux or HawkEye, but as Figure 4 shows, CBMM still sees a lower rate of long page faults than they do because its cost-awareness leads to fewer pathological cases, falling back to 4KB pages instead.

Even without fragmentation, CBMM always matches or improves on the tail latency of Linux and HawkEye, often by wide margins. In `xz`, `canneal`, and `memcached`, CBMM reduces the frequency of page faults taking 10-1000 μ s by two to three orders of magnitude compared to Linux or HawkEye. In `canneal` and `memcached`, CBMM reduces the frequency of (or eliminates) all page faults slower than 10 μ s by two or more orders of magnitude compared to Linux. In `memcached`, page faults taking over 1ms are nearly eliminated, while in `dc-mix`, they are reduced in frequency from nearly constant in Linux to every 10s or longer in CBMM. `mongodb` uses a userspace asynchronous I/O framework, as previously discussed, so its page fault latencies are dominated by context switches and other userspace threads; thus, our improvements are not visible in the figure. However, the figure does show that CBMM does not regress page fault latency, and as we will see in the next section, CBMM achieves significantly better performance than Linux or HawkEye for this workload.

Under fragmentation, CBMM usually achieves even larger tail latency improvements, particularly compared to Linux. For all workloads except `mongodb`, CBMM reduces the frequency of all page faults taking $\geq 50\mu$ s by 1-3 orders of magnitude compared to Linux and up to one order of magnitude compared to HawkEye. `mongodb` performs similarly to the unfragmented case, as discussed above.

Summary CBMM improves tail latency compared to Linux or HawkEye. For multiple workloads, CBMM reduces the frequency of slow page faults by one or more orders of magnitude, especially under fragmentation.

5.3 End-to-End Performance

CBMM’s major goal is to improve consistency and the debuggability of MM-related performance issues without degrading performance. Figure 5 shows the performance of each kernel with and without fragmentation. All results are normalized to Linux without fragmentation. Note that some of the performance difference of HawkEye compared to the other systems is due to Linux 4.3 (the black bar in the figure). On average, without fragmentation, CBMM has performance comparable to Linux and better than HawkEye. On average, with fragmentation, CBMM is 7% and 30% faster than HawkEye and Linux; in fact, it is only 12% slower than without fragmentation. With minimal tuning, on average, CBMM is 13% and 35% faster than HawkEye and Linux under fragmentation.

Without fragmentation, CBMM matches or exceeds the performance of Linux or HawkEye for all workloads except `canneal`. For `canneal`, CBMM is 15% slower than Linux because our profiles underestimate the benefit of huge pages. For `mongodb`, CBMM is 9% faster than Linux because it uses significantly more huge pages.

With fragmentation, CBMM outperforms Linux and/or HawkEye for all workloads except `mcf`. `mcf` uses too little memory to induce memory pressure; thus, CBMM overestimates the cost of huge pages and uses significantly fewer huge pages than Linux. In all other workloads, CBMM matches or outperforms at least one of Linux or HawkEye, often by wide margins. In `dc-mix`, `canneal`, and `memcached`, CBMM outperforms Linux by 34%, 34% and 81%, respectively, because its cost models allow it to adapt to a fragmented context, reflecting CBMM’s focus on consistent behavior. Notably, this includes all of our datacenter workloads.

To demonstrate CBMM’s benefit to performance debugging, we tune the performance of `mcf`, `canneal`, and `dc-mix` beyond the above results. In `mcf` and `dc-mix`, CBMM underestimates the benefit of huge pages, so we adjust the benefit upward in the respective profiles. We found that `canneal` exhibits a strong tradeoff between performance and page fault tail latency. As `canneal` is a non-interactive computational workload, we optimize for end-to-end performance by adjusting the profile to more aggressively allocate huge pages for the most important memory regions. After tuning, `dc-mix` without fragmentation runs 2% faster, and `mcf` with fragmentation runs 19% faster, than without tuning, but neither has a regression in tail latencies. `canneal` runs 18% faster than without tuning (46% faster than Linux) at the expense of some degradation in tail page fault latencies. In total, the tuning effort took less than a week, most of which was spent waiting for workloads to run.

Summary CBMM’s has competitive performance with Linux/THP and HawkEye and better tail latency and more interpretable behavior. In most cases, CBMM matches or exceeds Linux’s performance. Under fragmentation, CBMM often performs vastly better than Linux or HawkEye because of its

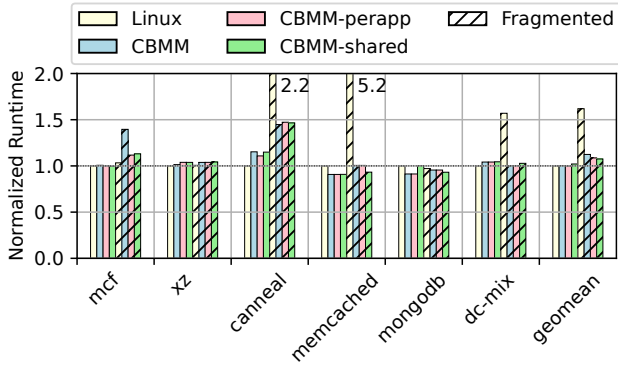


Figure 6: Runtime of CBMM workloads with generalized profiles, normalized to Linux with THP without fragmentations (lower is better).

focus on consistent behavior. Also, CBMM is easily debuggable and tunable by adjusting profiles and/or models.

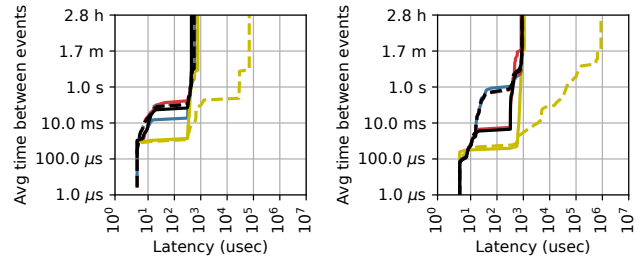
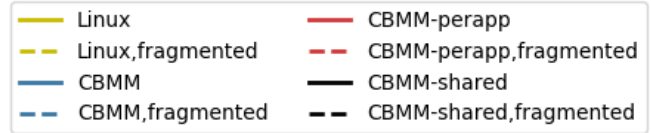
5.4 Efficiency

Allocating huge pages to memory regions that do not need them wastes contiguous memory and promotion overheads and possibly bloats memory usage. Figure ?? shows the percentage of anonymous memory used by each workload that is backed by a huge page in CBMM, HawkEye, and Linux (with THP) with and without fragmentation preconditioning.

Generally, preloaded profiles drive CBMM’s huge page usage, while HawkEye and Linux are more indiscriminate with huge page promotion. Usually, Linux attempts to use more huge pages than CBMM or HawkEye, often backing almost all memory with huge pages. HawkEye uses huge pages more efficiently than Linux, often achieving similar performance with much fewer huge pages. For most workloads, Linux still attempts to use huge pages under fragmentation, whereas CBMM and HawkEye do not, leading to significantly better tail latencies, and often better performance.

For `xz`, CBMM’s profile allows it to promote only a small but important part of the address space, so it matches Linux’s performance (and outperforms HawkEye) while using almost 80% fewer huge pages. For `mongodb`, CBMM outperforms Linux and HawkEye by using more huge pages in the absence of fragmentation and fewer in its presence, exemplifying CBMM’s cost-awareness.

Summary Despite having the most consistent behavior and sometimes better performance, CBMM often uses significantly fewer huge pages than Linux or HawkEye. By being cost- and context-aware, CBMM is more targeted in its use of huge pages, though in some cases, our profiles underestimate the benefit of huge pages.



(a) mcf

(b) xz

Figure 7: Soft page fault tail latency distribution weighted by page fault rate for different profiles. Compare to Figure 4.

5.5 Generality

CBMM has benefits even when a profile is highly imprecise, primarily by avoiding the pathological behavior of Linux. We compare three versions of profiles: the standard CBMM profile is as in Section 4.1, `perapp` assigns a single value to all memory regions in the workload equal to the average benefit of enabling THP for the workload, and `shared` is shared between all workloads and assigns a single value to all memory regions equal to the mean benefit of the `perapp` profiles.

Figure 7 shows how the different profiles affect page fault tail latency in `mcf` and `xz`. The `perapp` and `shared` profiles have minor regressions in page fault tail latencies compared to the standard profiles but still improve over Linux.

Figure 6 shows how the different profiles affect performance. In most cases, *CBMM with the simpler profiles outperformed Linux* with fragmentation, and the performance differences between the three profiles are within 5%. The `perapp` and `shared` profiles outperform the standard profiles slightly in some workloads. One exception is `mcf` under fragmentation, where both the `perapp` and `shared` profiles outperform the standard profile by 20%, similar to the tuned profile in Section 5.3, by being more liberal with huge pages.

Summary More precise profiles improve CBMM’s performance and tail latency, but imprecise profiles still have good results. Furthermore, profiles can be used to trade off performance and page fault latency.

5.6 CBMM Models

We evaluate the contribution of each model in Section 4 via three configurations of CBMM: `huge` enables only the huge page model, `async` additionally enables prezeroing, and standard CBMM enables all three models. Figure 8 shows the performance of these configurations, while Figure 9 shows tail latency for `mcf` and `xz`.

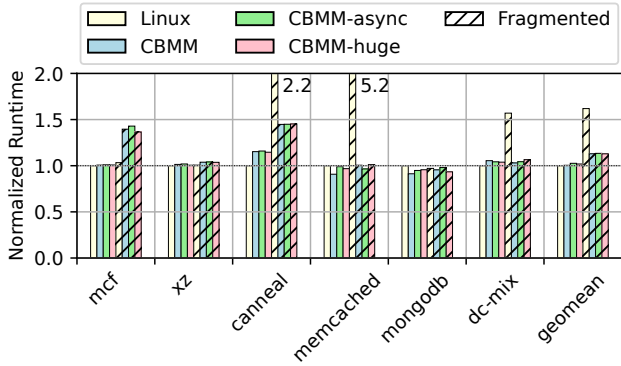


Figure 8: Runtime of CBMM workloads when enabling more models, normalized to Linux with THP without fragmentations (lower is better).

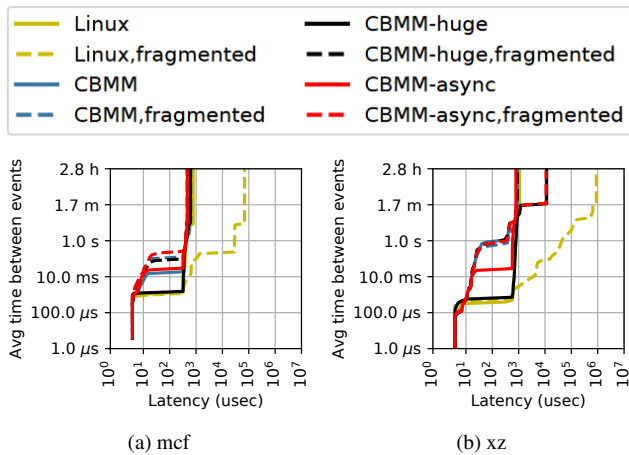


Figure 9: Soft page fault tail latency impact of different models. Compare to Figure 4.

Each policy provides benefits in different settings. The huge page model alone (*huge*) captures most of the performance benefit of CBMM because it prevents costly huge page allocations. Asynchronous prezeroing (*async*) decreases page fault tail latency by making huge pages cheaper. It also reduces performance slightly on an unfragmented system, where free pages abound, because prezeroing is wasted work. With fragmentation, prezeroing has little effect on performance.

Eager paging does not directly benefit performance but enables larger contiguous allocations where hardware supports it [29, 37, 40, 43, 49]. To evaluate how well CBMM can make large allocations, we compare the number of eagerly allocated regions and peak memory usage of CBMM with eager paging against Linux with `MAP_POPULATE`, the `mmap` flag that eagerly maps memory. In all workloads, regardless of fragmentation, CBMM uses eager paging for nearly the entire working set of the workload. Thanks to profiles, CBMM has $< 1\%$ memory bloat – 3%-48% less memory than `MAP_POPULATE` would use.

Summary CBMM’s huge page model provides significant tail latency (and often performance) improvements. Asynchronous prezeroing enables more huge page usage under

fragmentation, but has a modest cost on unfragmented systems. Eager paging has a modest performance cost but enables more contiguous memory allocation.

6 Related Work

Performance consistency at scale is a well-known problem [19] afflicting, among other systems, cluster computations [20] and distributed caching [11]. Redundancy is a common workaround [20]. MittOS uses deadline-aware kernel APIs to improve tail latency [26]. Like MittOS, we seek to fix consistency issues rather than mitigate their impact.

Kwon et al. observe that current huge page support is “a hodge-podge of best-effort algorithms and spot fixes” [31]. They and others identify real concerns and improve performance but often at the expense of increasing kernel heuristic complexity [14, 31, 38, 39, 49]. CBMM tames the increasing complexity of MM policy decisions by consolidating it in one place and reducing anomalous behavior.

VMware ESX Server explores MM techniques based on economic models by quantifying the value of idle memory and “taxing” processes for it [47]. Google and Meta both track and reclaim cold memory from processes, too [15, 32, 48]. Google’s system centrally and empirically coordinates content migration to far-memory tiers (e.g., compressed memory) [32], while Meta’s system relies on better metrics and acts locally on each machine. Google also profiles the lifetime of allocations to decrease memory fragmentation [35]. These approaches inspired our work; they use empirical measurements and MM-wide guiding principles to make MM decisions. Our work extends and generalizes this idea. Sriraman et al. take a step in this direction by comprehensively profiling Meta’s workloads and using the profiles to guide coarse-grained boot-time system tuning [42].

There is much prior work on asynchronous prezeroing of pages [8, 21, 22, 33, 38, 46]. Recent work observes that larger page sizes and non-temporal store instructions make prezeroing useful again [38]. We demonstrate the usefulness of our approach by quantifying zeroing costs and the prezeroing implementation, and integrating them into our prototype.

7 Conclusion

Modern computing needs are placing new demands on kernel MM. To meet these demands, kernel MM must begin to prioritize behavioral consistency and debuggability. We propose CBMM, a MM system that uses cost-benefit analysis to make policy decisions. Despite using relatively simple models in its cost-benefit estimation, CBMM’s principled approach to MM allows matching the performance of existing systems while also improving system behavioral consistency. CBMM paves a way for kernel MM behavior to become less opaque, unlocking further performance and optimizations in the future.

Acknowledgements

We thank the anonymous reviewers, Sujay Yadalam, and Yuvraj Patel for their time and insightful feedback on our paper. We thank the anonymous artifact reviewers and Anthony Rebello for their time spent testing our artifact. We thank Ashish Panwar for the help getting HawkEye set up. We thank Michael Marty who gave feedback on early versions of the project that became CBMM.

This work was funded by NSF grants CNS 1815656 and CNS 1900758.

Availability

Our artifact is open-source and available at <https://github.com/multifacet/cbmm-artifact>. See Appendix A for further details.

References

- [1] Borg Cluster Traces from Google. <https://github.com/google/cluster-data>.
- [2] Database Installation Guide. https://docs.oracle.com/cd/E11882_01/install.112/e47689/pre_install.htm#LADBI1152.
- [3] Disable Transparent Huge Pages (THP). <https://docs.mongodb.com/manual/tutorial/transparent-huge-pages>.
- [4] Disabling Transparent Huge Pages (THP). <https://docs.couchbase.com/server/current/install/thp-disable.html>.
- [5] SPEC CPU 2017 Benchmark Suite. <https://www.spec.org/cpu2017/>.
- [6] Microsoft Azure Traces. <https://github.com/Azure/AzurePublicDataset>.
- [7] Redis Latency Problems Troubleshooting. <https://redis.io/topics/latency>.
- [8] Remove PG_ZERO and zeroidle (page-zeroing) entirely. <https://news.ycombinator.com/item?id=12227874>, August 2016.
- [9] Vlastimil Babka. [Overview of memory reclaim in the current upstream kernel](#). In *Linux Plumbers Conference 2021*, September 2021.
- [10] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Morgan & Claypool Publishers, 2013.
- [11] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Mor Harchol-Balter, and Siddhartha Sen. [RobinHood: Tail Latency-aware Caching – Dynamic Reallocating from Cache-rich to Cache-poor](#). In *Proceedings of the Thirteenth Conference on Operating Systems Design and Implementation*, OSDI, 2018.
- [12] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O’Reilly Media, Inc., 1st edition, 2016.
- [13] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.
- [14] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. [Theseus: an Experiment in Operating System Structure and State Management](#). In *Proceedings of the Fourteenth Symposium on Operating Systems Design and Implementation*, OSDI, 2020.
- [15] Shakeel Butt, Suren Baghdasaryan, and Yu Zhao. [Finding more DRAM](#). In *Linux Plumbers Conference 2019*, 2019.
- [16] Richard W. Carr and John L. Hennessy. [WSCLOCK – a Simple and Effective Algorithm for Virtual Memory Management](#). In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP, 1981.
- [17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. [Benchmarking Cloud Serving Systems with YCSB](#). In *Proceedings of the First ACM Symposium on Cloud Computing*, SoCC, 2010.
- [18] Dan Williams. Randomize free memory. <https://lwn.net/Articles/767614/>.
- [19] Jeffrey Dean and Luiz André Barroso. [The tail at scale](#). *Communications of the ACM*, 56(2), February 2013.
- [20] Jeffrey Dean and Sanjay Ghemawat. [MapReduce: Simplified Data Processing on Large Clusters](#). In *Proceedings of the Sixth Conference on Symposium on Operating Systems Design & Implementation*, OSDI, 2004.
- [21] Cort Dougan, Paul Mackerras, and Victor Yodaiken. [Optimizing the idle task and other MMU tricks](#). In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI, 1999.
- [22] Lars Eggert, Alan Cox, Cort Dougan, and Matt Dillon. [Clearing Pages in the Idle Loop](#). <https://www.mail-archive.com/frebsd-hackers@frebsd.org/msg13993.html>, July 2000.
- [23] Jason Evans. [Scalable memory allocation using jemalloc](#), January 2011.

- [24] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. **BadgerTrap: a Tool to Instrument x86-64 TLB Misses**. *ACM SIGARCH Computer Architecture News*, 42(2), 2014.
- [25] Mel Gorman and Andy Whitcroft. **The What, the Why and the Where to of Anti-fragmentation**. In *Proceedings of the Linux Symposium*, volume 1, January 2006.
- [26] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Christina Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. **MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface**. In *Proceedings of the Twenty-Sixth Symposium on Operating Systems Principles*, SOSP, 2017.
- [27] John Wilkes. **More Google Cluster Data**. <http://ai.googleblog.com/2011/11/more-google-cluster-data.html>.
- [28] Frans Kaashoek, Robert Morris, and Yandong Mao. **Optimizing MapReduce for Multicore Architectures**. Technical Report MIT-CSAIL-TR-2010-020, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 2010.
- [29] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. **Redundant Memory Mappings for Fast Access to Large Memories**. In *Proceedings of the Forty-Second Annual International Symposium on Computer Architecture*, ISCA, 2015.
- [30] Paul Kocher, Jann Horn, Anders Fogh, and Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. **Spectre attacks: Exploiting speculative execution**. In *Fortieth IEEE Symposium on Security and Privacy*, S&P, 2019.
- [31] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. **Coordinated and Efficient Huge Page Management with Ingens**. In *Proceedings of the Twelfth Conference on Operating Systems Design and Implementation*, OSDI, 2016.
- [32] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhail, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. **Software-Defined Far Memory in Warehouse-Scale Computers**. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2019.
- [33] Christopher Lameter. **Increase page fault rate by prezeroing V1 [0/3]: Overview**. <https://lkml.org/lkml/2004/12/21/142>, December 2004.
- [34] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. **Meltdown: Reading kernel memory from user space**. In *Twenty-Seventh USENIX Security Symposium*, USENIX Security, 2018.
- [35] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. **Learning-based Memory Allocation for C++ Server Workloads**. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2020.
- [36] Mark Mansi and Michael M. Swift. **Osim: Preparing System Software for a World with Terabyte-scale Memories**. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2020.
- [37] Michal Nazarewicz. **A deep dive into CMA**. <https://lwn.net/Articles/486301/>.
- [38] Ashish Panwar, Sorav Bansal, and K. Gopinath. **Hawk-Eye: Efficient Fine-grained OS Support for Huge Pages**. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2019.
- [39] Ashish Panwar, Aravinda Prasad, and K. Gopinath. **Making Huge Pages Actually Useful**. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2018.
- [40] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. **Large pages and lightweight memory management in virtualized environments: can you have it both ways?** In *Proceedings of the Forty-Eighth International Symposium on Microarchitecture*, MICRO-48, 2015.
- [41] Robert Ricci, Eric Eide, and CloudLab Team. **Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications**. *login.*, 39(6), December 2014.
- [42] A. Sriraman, A. Dhanotia, and T. F. Wenisch. **SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale**. In *Proceedings of the Forty-Sixth Annual International Symposium on Computer Architecture*, ISCA, 2019.

- [43] Madhusudhan Talluri and Mark D. Hill. [Surpassing the TLB performance of superpages with less operating system support](#). *ACM SIGPLAN Notices*, 29(11), November 1994.
- [44] Huangshi Tian, Yunchuan Zheng, and Wei Wang. [Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud](#). In *Proceedings of the ACM Symposium on Cloud Computing, SoCC*, 2019.
- [45] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. [Borg: the Next Generation](#). In *Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys*, 2020.
- [46] Linus Torvalds. [Page zeroing strategy](#). https://yarchive.net/comp/linux/page_zeroing_strategy.html, December 2000.
- [47] Carl A. Waldspurger. [Memory Resource Management in VMware ESX Server](#). In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, OSDI*, 2002.
- [48] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. [TMO: transparent memory offloading in datacenters](#). In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2022.
- [49] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. [Translation Ranger: Operating System Support for Contiguity-aware TLBs](#). In *Proceedings of the Forty-Sixth International Symposium on Computer Architecture, ISCA*, 2019.
- [50] Wenbo Zhang. [Why We Disable Linux's THP Feature for Databases](#). <https://pingcap.com/blog/why-we-disable-linux-thp-feature-for-databases>, December 2020.
- [51] Weixi Zhu, Alan L. Cox, and Scott Rixner. [A Comprehensive Analysis of Superpage Management Mechanisms and Policies](#). In *2020 USENIX Annual Technical Conference, ATC*, 2020.

A Artifact Appendix

Abstract

In order to aid future research and facilitate the reproduction of our work, we open-sourced our artifact, which is available at <https://github.com/multifacet/cbmm-artifact>. Our artifact includes both the CBMM kernel, which is a modification of the 5.5.8 Linux kernel, and our tooling for running the experiments discussed in the paper. The [README.md](#) file in the artifact contains detailed instructions for running each experiment in the paper and reproducing the results and plots therein.

Scope

Running the experiments as specified in the [README](#) on similar hardware to our own setup (described in Section 5.1) should allow the reviewer to generate comparable results to those in the accepted version of the paper.

Specifically, our paper's key claims are:

- CBMM improves page fault tail latency, our measure of MM system behavioral consistency, compared to Linux and HawkEye (Figures 2 and 4).
- CBMM does not regress application runtime, and under fragmentation can often significantly improve runtime compared to Linux and/or HawkEye (Figure 5).
- CBMM often uses huge pages more frugally than Linux or HawkEye despite getting better tail latency and comparable (or better) performance (Figure 6).
- CBMM has benefits even when profiles are imprecise (Section 5.5, 5.6).

Because running all experiments is time and resource intensive, we provide a screencast and intermediate results for the reviewers. This should allow generation of checkable partial results in a reasonable amount of time.

Contents

This artifact contains:

- [README.md](#): contains instructions for how to use the artifact.
- [paper.pdf](#): the accepted version of the paper, without any modifications responding to reviewer requests.
- [cbmm/](#): a git submodule containing our primary artifact, the CBMM kernel, which is a modified version of Linux 5.5.8.
- [cbmm-runner/](#): a git submodule of our runner tool, which runs our experiments.

- [profiles/](#): a set of profiles we used in our evaluation. More info is available in the [README](#).
- [scripts/](#):
 - Convenience scripts for running experiments (more in "Detailed Instructions"),
 - Scripts for processing experimental output into a consumable/plottable form,
 - Scripts for plotting experimental results to generate the figures from the paper.
- [figures/](#): copies of the figures from the paper.

Hosting

Our artifact is hosted on GitHub at <https://github.com/multifacet/cbmm-artifact>. Git tag [atc22ae](#) specifies the version submitted for review, but more recent versions of the [main branch](#) contain helpful additions, such as additional figures not included in the paper for lack of space.

Requirements

Reviewers will need a machine with specs similar to Section 5.1:

- 192GB DRAM
- Multiple cores
- \geq 50GB free disk space
- Running Centos 7
- Can install the CBMM kernel in place of the existing Linux kernel
- Internet connection

They will also need any other Linux machine that can connect to the first machine via passwordless SSH. This machine drives the experiments to run on the first machine.

They will also need access to SPEC 2017 ISO, which we cannot provide due to licensing constraints.

Full details are in the [README](#).



EPK: Scalable and Efficient Memory Protection Keys

Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, Haibo Chen

Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Institute of Parallel and Distributed Systems (IPADS), SEIEE, Shanghai Jiao Tong University

Abstract

As a hardware mechanism for facilitating intra-process memory isolation, Intel Memory Protection Keys (MPK) has been leveraged to efficiently improve the isolation, security, or performance of the software. However, it can only support 16 isolated memory domains, which significantly limits its applicability in many scenarios.

In this paper, we present EPK which leverages off-the-shelf virtualization hardware features to extend the number of available protection domains in MPK. To demonstrate the effectiveness of EPK, we apply it in three scenarios, including better memory isolation for server applications as well as Non-Volatile Memory (NVM) applications, and a fast Inter-Process Communication (IPC) mechanism for microkernels. The evaluation results show that EPK can scale to provide hundreds of isolated domains. It can outperform the state-of-the-art (libmpk) by up to two orders of magnitude and usually achieve 95% of the performance of the system with no memory isolation.

1 Introduction

Intel MPK [7] has attracted many researchers since introduced in 2019 because it offers highly-efficient intra-process memory isolation by supporting memory domains inside one application. An application can switch between different domains with a new instruction, *WRPKRU*, which can execute in the user mode directly and takes only about 28 cycles. Compared with traditional software isolation or page table based isolation, MPK can achieve much lower performance overhead, and has been adopted in many scenarios, including: 1) enhancing the isolation between different threads of the same process by giving them different domain views [13, 55, 57]; 2) hardening the security of an application by separating different components, such as untrusted third-party libraries, into different domains [24, 40, 44, 46, 50]; and 3) improving the performance of software that uses multiple page tables for isolation by substituting domains for page tables [22, 29].

However, the small number (16) of isolated memory domains supported by MPK severely undermines its usability. First, typical server applications usually serve for more than 16 clients concurrently, and it is preferable to store clients' private data in isolated domains to prevent sensitive data leakage due to vulnerabilities like Heartbleed [5]. Second, there is a growing interest in protecting persistent memory [6] data from accidental or malicious accesses [56, 57]. Long-lived persistent data is usually directly mapped into processes and then accessed via load/store instructions. Isolating the data in more domains can reduce the data exposure time and benefit stray access protection. Third, both applications and system software may contain more than 16 components that need to be isolated. For example, popular applications use scores of third-party libraries [2]; an OS consists of tens or hundreds of modules like device drivers. Besides, prior studies also indicate the performance of NVM applications (which desire isolation) [57] and microkernel OSES [22] can boost by more than 10× with more MPK domains.

To scale MPK beyond 16 memory domains, recent researchers propose either software or hardware approaches to support more MPK domains [38, 41, 57]. However, the software approach suffers from a large overhead while the hardware approaches are infeasible on commodity machines.

In this paper, we propose EPK, which extends the maximum number of memory domains supported by MPK on commodity hardware efficiently. MPK's performance advantage stems from the decoupling of domain configuration (in privilege mode) and domain switching (in non-privilege mode). Our observation is that another hardware feature, named *fast EPT-switching* (Extended Page Table switching, with *VMFUNC*), has a similar pattern, which decouples EPT configuration (in host mode) from EPT switching (in guest mode). Thus, we propose *extended protection keys* by combining MPK with fast EPT-switching, i.e., reusing the same MPK protection keys in different extended page tables (EPT). Thus, with 512 EPTs, EPK can support up to 7,680 domains.

However, there are two major challenges to the new system. The first challenge is to provide a unified abstraction for appli-

cations although combing two orthogonal hardware features. EPK still retains the abstractions of memory domain and domain switching inherited from MPK while hiding the EPTs from applications, by elaborately managing domain mappings in multiple EPTs and developing a library to provide easy-to-use APIs. The second challenge is to enable one thread to simultaneously access memory domains across different EPTs, as the original MPK allows to access multiple domains together. To this end, EPK leverages another existing hardware feature named virtualization exception (VE) to switch the EPTs for the thread transparently when a domain access causes EPT violations.

We implement EPK prototype and apply it in the above scenarios. On the one hand, EPK can work like the original MPK for mitigating the memory errors and thus facilitates efficient intra-process memory isolation (Section 5). On the other hand, it can also isolate untrusted software components [46, 57] (Section 4 and Section 6) by further preventing illegal domain switching.

Experiments on server applications and persistent memory applications show that EPK’s overhead is usually around or below 5%. Compared with the state-of-the-art (libmpk) [38], the performance improvement can be up to two orders of magnitude. Furthermore, we incorporate EPK in a microkernel OS, a representative of large software. A microkernel OS runs system components like file systems and device drivers in user processes for embracing better isolation [20, 26, 30]. Nevertheless, costly inter-process communication (IPC) is required for the interaction between different OS components [22, 31, 37, 45]. EPK can provide enough isolated domains for running different OS components and the fast domain switch for IPCs. Thus, we propose a high-efficient IPC mechanism named HyBridge that can improve the performance of three well-known microkernels, seL4 [10], Google Zircon [4], and Fiasco.OC [3], and outperform two state-of-the-art IPC designs, SkyBridge [37] and UnderBridge [22].

In summary, this paper makes the following contributions: 1) a scalable and efficient intra-process memory isolation mechanism named EPK; 2) a real implementation and evaluation on Linux; 3) a new IPC design based on EPK for microkernel OSes with better performance.

2 Background and Motivation

2.1 Hardware Background

MPK. Intel MPK [7] can divide the virtual memory space of one process into 16 memory domains. By leveraging previously unused bits of the page table entry, each memory page is tagged with a four-bit protection key as the domain ID and exclusively belongs to one of the 16 domains. A new 32-bit register, *PKRU*, is introduced to specify the access permissions (read-only, read-write, none) on the 16 domains (two bits for one domain). Because the register is per-core, con-

current threads in the same process can have different access permissions on different domains. During runtime, MMU transparently checks the permissions. A non-privileged instruction called *WRPKRU* can update this register to change the access permissions.

MPK in the VM. The hardware feature of MPK is also usable in a VM. Protection keys are still tagged in the page tables of applications instead of EPTs. From the perspective of applications and the OS, the usage of MPK is just the same no matter in the VM or not.

Extended Page Table (EPT) and VMFUNC. Intel hardware virtualization technology employs EPT for memory virtualization. For a guest virtual machine (VM), the guest page table maps guest virtual addresses (GVA) to guest physical addresses (GPA) while the EPT maps GPAs to host physical addresses (HPA) and thus aids in the seamless translation of GVAs to HPAs. The guest VM’s OS (runs in non-root mode ring zero) controls the guest page table, while the hypervisor (runs in root mode) manages the VM’s EPT. *VMFUNC* is a hardware virtualization extension that provides VM functions for VMs. EPT pointer (EPTP) switching is currently the only VM function provided, allowing the guest VM (both Ring-0 and Ring-3) to directly load a new EPTP. The loadable EPTP can only be chosen from a list of EPTPs (up to 512) configured by the hypervisor. Note that TLB entries are tagged with the EPT base addresses to avoid flushing the TLB when switching the EPT.

Virtualization Exceptions (VE). EPT violations usually trigger VMExits, after which the hypervisor can fill the EPT mappings. Yet, Intel virtualization technology also supports converting EPT violations into VE without VMExits. With VE enabled, the hypervisor can configure bit 63 of certain EPT paging-structure entries to make EPT violations on some GPAs to cause VE and others to cause VMExits as before.

2.2 Motivation

Software fault isolation (SFI) can enhance memory isolation for applications [15, 19, 27, 36, 48, 58] by instrumenting and restricting memory accesses. Nonetheless, it may result in non-negligible runtime performance overhead and is inflexible (e.g., hard to be fine-grained). Many studies can avoid such disadvantages [25, 27, 32, 35, 39] using the MMU. They isolate different memory partitions of a process in different page tables or extended page tables and thus utilize MMU to check memory accesses at the page granularity.

Instruction	Cost (cycles)	Solution	Overhead
Write <i>CR3</i> (no TLB flush)	226	<i>LwC-simulate</i>	70%
<i>VMFUNC</i> (switch EPT)	146	<i>EPT-based</i>	12%
<i>WRPKRU</i>	28	<i>ERIM</i>	3%

(a)

(b)

Table 1: (a) Instruction cost. (b) The overhead of isolating session keys in one isolated domain.

However, constructing different memory domains with page tables is not free. Switching between different domains requires changing the page table through specialized hardware instructions. Table 1(a) presents the direct cost of the related instructions. We design an experiment to isolate each client’s session key in separate domains in the NGINX web server [9] to show the corresponding performance overhead. ApacheBench (ab) [1] generates the workload: 300 concurrent clients send requests to the server for a file. As presented in Table 1(b), light-weight contexts (lwC) [32], as a representative of page-table-based solutions, will lead to approximately 70% overhead if we isolate all the session keys in a separate context (i.e., a new page table) and switch to that context when accessing those keys. Similarly, for the EPT-based solution, we create a new EPT for isolating all the session keys and use *VMFUNC* instruction to switch to that EPT when accessing them. Although noticeably better than the page-table-based solution, such an EPT-based solution still introduces around 12% performance overhead. In contrast, ERIM [46] only adds about 3% overhead by utilizing MPK to construct an isolated memory domain for storing the session keys, which can demonstrate the efficiency of MPK.

Yet, MPK can only support at most 16 memory domains, limiting its usage. Take the web server for example: it is preferable to separate clients’ data in different memory domains, guaranteeing the isolation between multiple clients. Recent work [38, 41, 57] also identifies and addresses this limitation of MPK. Two studies [41, 57] propose non-trivial hardware extensions for efficiently supporting scalable domains, which are not achievable on current platforms.

libmpk [38] gives the illusion of multiple memory domains by exposing virtual keys to applications and maintaining the mapping between virtual keys and the 16 real keys (one key for one domain). When all 16 real keys are exhausted and a new virtual key is required, libmpk will evict a mapped real key and remap it to the new virtual key. But the key eviction may incur a large overhead. For instance, if we protect each client’s session key in a different memory domain (300 domains in total) provided by libmpk in the above NGINX experiment (rather than storing all keys in one domain), the overhead becomes about 20%. The overhead consists of both direct costs, i.e., the expensive key eviction procedure involving modifying page table entries, flushing TLBs, etc., and indirect costs, e.g., TLB misses due to flushing.

More seriously, libmpk’s domain switch cost increases as domain memory gets larger, as shown in Table 2. The micro-benchmark keeps switching to one domain randomly. When the domain number increases from 32 to 64, more key eviction occurs, resulting in higher overhead. As one domain contains more memory pages, the switch cost gets more expensive due to flushing more TLBs and updating more page table entries. The cost turning point (from 33 to 34) is because Linux flushes all TLBs together instead of one at a time when the number of TLBs to flush exceeds 33.

Domains \ Pages	16	33	34	64	1K	128K
15	185	184	188	188	187	185
32	6,576	11,173	4,090	5,270	42,912	5.1×10 ⁶
64	9,959	16,573	6,308	8,068	79,012	9.6×10 ⁶

Table 2: The CPU cycles of domain switches in libmpk. The page size is 4k.

	Memory Access Cost	Domain Switch	Memory Domain Number	Multi-domain Access	Multi-thread Support	Hardware Changes
SFI	High	Fast	Many	No	Yes	Zero
lwC	Low	Slow	Many	No	Yes	Zero
Donkey	Low	Fast	1,024	Yes	Yes	Heavy
libmpk	Low	Slow	Many	No	No	Zero
MPK	Low	Fast	16	Yes	Yes	Zero
EPK	Low	Fast	7,680	Yes	Yes	Zero

Table 3: Comparison of different approaches.

In brief, MPK-based intra-process memory isolation shows attractive performance advantages but can only support a limited number of isolated domains. Therefore, we intend to overcome this limitation while retaining MPK’s performance and flexibility advantages. As described in Table 3, SFI-based and page-table-based approaches (e.g., lwC) have performance issues and do not allow one thread to simultaneously access different domains. Existing hardware approaches (e.g., Donkey [41]) are hard to be implemented on commercial x86/ARM architectures due to intrusive hardware modifications. For example, to support 1024 domains, Donkey takes 10 bits in the page table entry as the domain ID, which is at least incompatible with the upcoming 5-level page table. libmpk makes several contributions like implementing fast *mprotect* by using MPK. But, its extension on the MPK domain number has both performance and flexibility issues. It cannot support multi-threading well, in particular, because it is difficult to maintain a consistent view of active domains across different threads.

3 The EPK Mechanism

According to prior studies on MPK-based intra-process isolation, the common usage model of MPK is as follows. An application (process) creates memory domains by binding different protection keys (pkey) to them as the domain IDs and separates the memory data into different domains. An application thread acquires/releases the access permission of one specific domain before/after accessing the data in it, which reduces the chances of the isolated memory being affected by vulnerabilities (e.g., leakage caused by buffer overflow) or faults (e.g., wild writes). Acquiring the domain access permis-

sion is efficiently achieved by executing *WRPKRU* instruction, which is referred to as switching to that domain. Releasing the permission is a reverse procedure that also makes use of *WRPKRU*. EPK still inherits such a usage model while supporting more memory domains.

Extended Protection Keys. The root cause of why MPK can only support 16 memory domains for one application is that each domain needs to exclusively take one pkey while the hardware only supports 16 pkeys. So, to extend the number of memory domains, the high-level idea of EPK is allowing multiple memory domains to use the same pkey at the same time. However, simply reusing the same pkey for different domains does not guarantee memory isolation. Therefore, EPK proposes *extended protection key*, which extends a pkey with different EPT indexes (get more keys), and then assigns different extended protection keys to different memory domains.

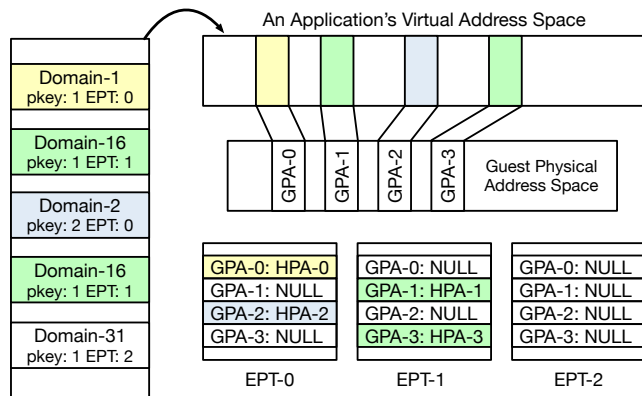


Figure 1: The memory mapping overview for an application.

As depicted in Figure 1, EPK allows an application to partition its virtual address space into different memory domains, with each domain containing discrete memory pages. A domain exclusively takes one *extended protection key* as its domain ID, which is composed of a pkey (1-15) and an EPT index (0-N, $0 \leq N < 512$)¹. EPK requires an application to run within a VM where cloud applications usually run in, and multiple EPTs need to be created for the VM. Each EPT can hold 15 domains for an application (domain-0 is used as the shared domain), and the 15 domain IDs (extended protection keys) have the same EPT index but different pkeys. For example, domain-1 and domain-2 are both in EPT-0 and use pkey-1 and pkey-2, respectively. The same pkey can be shared by domains in different EPTs concurrently, e.g., domain-1, domain-16, and domain-31 can all use pkey-1 because they will be mapped in EPT-0, EPT-1, and EPT-2, separately. Memory isolation between domains within the same EPT is achieved through the use of distinct pkeys. To achieve memory isolation between domains in different EPTs, EPK ensures that each domain's mappings only exist in one EPT. Specifically, the memory pages belonging to one isolated memory domain

¹ Domain-ID (extended protection key) = EPT-index \times 15 + pkey.

are tagged with the domain's pkey in the application's page table and are only mapped in the domain's EPT. Other memory pages, i.e., the global code and data of an application, are tagged with pkey-0 and mapped in all the EPTs (domain-0).

Although all the 512 EPTs are shared among different applications, it is worth mentioning that each application can construct 7,680 domains (15×512) since it has an individual guest page table.

Domain Switching. When an application thread needs to access some domain, it retrieves the permission by setting the PKRU value and choosing the corresponding EPT (switching to the domain). Switching between domains within the same EPT can be finished by executing one *WRPKRU* instruction. Switching between domains in different EPTs involves one additional *VMFUNC* instruction for EPT switching. Since both these two instructions are non-privileged, the domain switches are efficiently finished in user mode (one exception case will be explained in Section 3.2). From the perspective of programming, EPK provides easy-to-use interfaces (Section 3.3) through a user-level library for applications to create/delete domains, add/remove memory pages to/from domains, and switch domains. Applications can simply use the interfaces similar as programming on the original MPK.

Challenges. Although the idea sounds simple, there are two implementation challenges for combining the hardware features. First, how to make a VM seamlessly run with different EPTs, and how to differentiate a legal EPT violation caused by on-demand domain paging with an illegal one due to an unauthorized access? (Section 3.1). Second, given that MPK allows one thread to access multiple domains simultaneously, how to support such a flexible feature when multiple EPTs are in use (access domains mapped in different EPTs simultaneously)? (Section 3.2).

Threat Model. We assume the guest OS, hypervisor, and hardware are trusted, and EPK is correctly implemented. For the case of reducing the memory exposure time (Section 4), we assume the unreliable code may contain memory corruption bugs, which is similar to the existing work [38, 57]. For the case of isolating mutual-distrusted software components (Section 5 and (Section 6)), we assume the untrusted code or mutual-distrusted code may contain exploitable vulnerabilities like memory corruption and even use ROP to abuse *WRPKRU/VMFUNC* for illegal domain switches. So, EPK further integrates the mechanism of secure switching from previous systems [22, 46] (Section 6.1 explains how to avoid illegal domain switches). Other attacks, like side-channel attacks and rowhammer attacks, are not considered.

3.1 Extended Page Table Management

Traditionally, a VM has a single EPT that maps the GPAs of both the guest OS and applications to HPAs. Differently, EPK necessitates the creation of multiple EPTs for a VM based on two principles. *Principle-1:* GPAs that are not allocated for

memory domains should be mapped uniformly across EPTs. Thus, the VM can always run normally in any EPT. *Principle-2*: Each memory domain's GPAs should be mapped in only one EPT. As previously stated, this is for domain isolation.

GPA to Domain Association. Since the hypervisor is in charge of constructing EPTs, the first problem is how it can tell whether one GPA belongs to some memory domain or not. A straightforward solution is letting the guest OS, the GPA manager, share the information about which GPAs are allocated for memory domains with the hypervisor. Nevertheless, this entails non-trivial modifications to both the hypervisor and the guest OS. An alternative solution is to divide the whole GPA space into two halves and allocate GPAs for domain memory from one half, allowing the hypervisor to easily determine whether a GPA belongs to a domain. This solution still adds a significant amount of complexity to GPA allocation in the guest OS.

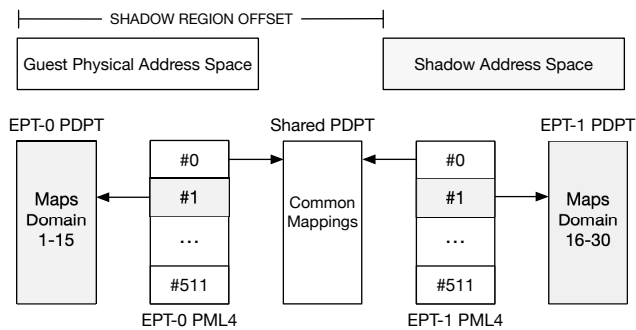


Figure 2: The EPT structures. PML4 is the top-level EPT page, and PDPT is the second top-level page.

To address this problem, EPK proposes the following design. Instead of partitioning the GPA space, EPK creates the illusion that there is a shadow address space (GPA) in the VM by simply adding a fixed offset (*SHADOW REGION OFFSET*) to the GPAs allocated to memory domains, as illustrated in the top half of Figure 2. As a result, the fixed offset becomes the boundary between the GPAs for memory domains and other GPAs. Based on this boundary, EPK constructs the EPTs, as shown in Figure 2. It sets the offset to 512 GB since an EPT PML4 entry can point to 512 GB GPA range. The entire GPA space is pointed by the first entry of each EPT PML4², and the shadow address space is pointed by the second entry of each EPT PML4. The first PML4 entry of different EPTs points to a shared PDPT, implying that the non-domain GPA mappings are always the same in different EPTs and thus satisfies *Principle-1*. By sharing this PDPT, the hypervisor can reduce the space overhead of multiple EPTs. More importantly, it does not need to explicitly synchronize an EPT update (e.g., adding a new mapping for the guest OS) across all EPTs, which is expensive. The second PML4 entry

²For simplicity, we assume the size of the GPA space is smaller than 512 GB. The fixed offset can be adjusted to support larger GPA space.

of different EPTs points to different PDPTs for adding the GPA mappings for memory domains, which is a prerequisite of *Principle-2*.

Illegal EPT Violations. The second problem is the hypervisor cannot determine if an EPT violation (EPT fault) within the shadow address space is legal or not. Assume an application thread executes in EPT-1 while accessing Domain-1 in EPT-0, resulting in an EPT violation. The hypervisor cannot decide whether to add the mapping because it does not know which domain the faulting address belongs to, i.e., whether the GPA should be mapped in the current EPT. Simply adding the mapping regardless of semantics will violate *Principle-2*. Instead, EPK chooses to avoid any legal EPT violation within the shadow address space (except accessing domains across EPTs which will be explained in Section 3.2). Specifically, the guest OS is required to invoke one new hypercall (a hypervisor interface provided to the VM) to fill the EPT mapping when a legal domain page fault happens, which eliminates the following EPT violation. The guest OS can check the legality of a domain page fault because applications tell it the semantics of domain mappings via the corresponding interfaces (explained in Section 3.3). As such, the hypervisor only needs to add a simple hypercall to add the EPT mapping, and EPT violations within the shadow address space must be illegal. Together with the carefully designed EPT structure, *Principle-2* can be met now. Furthermore, because it avoids original VMExits caused by EPT violations, this hypercall-based solution incurs no additional overhead.

EPT-ID Access. When a domain page fault occurs, the guest OS needs to check whether the faulting thread has the access permission according to the current PKRU value and EPT-ID. However, because the domain switches are performed in user mode, the guest OS is unaware of the changes of PKRU and EPT-ID. The guest OS can directly read the PKRU register but cannot get EPT-ID (the third problem). EPK enables the guest OS to efficiently retrieve the EPT-ID by subtly mapping one special guest physical page (named EPT-ID-Page) across different EPTs. During VM initialization, the guest OS allocates the EPT-ID-Page and passes its address to the hypervisor. The hypervisor maps the EPT-ID-Page to different host physical pages in different EPTs (in different PDPTs) and stores the corresponding EPT-ID in each physical page. Therefore, the guest OS can always obtain the current EPT-ID by simply reading the EPT-ID-Page (first four bytes).

3.2 Multi-Domain Access Support

MPK supports 16 domains and allows one thread to access any of them by configuring the PKRU register. Nevertheless, it is non-trivial to support this flexible feature in EPK since there are domains across different EPTs.

Accessing multiple domains in the same EPT can still be accomplished simply by configuring PKRU. To transparently support accessing multiple domains in different EPTs, EPK

further employs another hardware feature named VE (virtualization exception). The hypervisor converts EPT violations in the shadow address space into VEs which will be handled in the guest OS. The VE handler in the OS can switch the EPTs for one thread and thus help it to seamlessly access multiple EPTs. Specifically, when a thread needs to acquire the access permission of domains across multiple EPTs simultaneously, it needs to inform the kernel of the domain information. Suppose the thread needs to access domain-A in EPT-1 and domain-B in EPT-2 and first runs in EPT-1. As running in EPT-1, it can directly access domain-A but will trigger an EPT violation when attempting to access domain-B. Because domain-B is in the shadow address space, the corresponding EPT violation will be caught by the VE handler instead of causing expensive VMExits. Since the OS knows that the thread can access domain-B, the VE handler will switch to EPT-2 by using VMFUNC and setting PKRU to the required value. After that, the thread can be restored and continue to access domain-B. A similar procedure happens when it later accesses domain-A in EPT-2. Thereby, EPK gives an illusion that one thread can access domains in multiple EPTs at the same time.

Two points are worth mentioning. First, EPK only converts EPT violations to VEs within the shadow address space, which has no interference on the VM's original execution. Second, different from getting access to domains in the same EPT or a specific domain in other EPTs (fast path), getting access to multiple domains in different EPTs requires the kernel involvement (slow path).

3.3 System Components in Linux/KVM

EPK's prototype implementation on Linux/KVM mainly consists of three components: a user library, a kernel module in the guest OS (Linux), and a hypercall handler in the hypervisor (KVM).

Figure 3 lists the main library interfaces available to applications. The first two functions invoke the kernel module through *ioctl* to allocate and free domain IDs. *alloc_domains* can get multiple domain IDs, and the kernel module will try

```

/* Allocate domain IDs with affinity */
int alloc_domains(int num, int dom_ids[]);

/* Free domain IDs */
int free_domains(int num, int dom_ids[]);

/* Allocate a virtual memory range for a domain */
void *domain_mmap(int dom_id, void *addr, size_t len,
                 int prot, int flags);

/* Remove some mappings */
int domain_munmap(void *addr, size_t len);

/* Retrieve the access permission of a domain */
int domain_begin(int id, int prot);

/* Release the domain permission */
int domain_end(int id);

```

Figure 3: The APIs provided by the user library of EPK.

to return the domains that are located in the same EPT. This is because some domains may have affinities, i.e., they are likely to be traversed together. Properly utilizing affinity in the applications can benefit the performance. Although it is non-trivial in general, achieving locality is straight forward in some cases. For example, in Section 4.2, a simple locality-aware request dispatching scheme can make Memcached embrace the affinity benefits; in Section 5.2, simply letting one thread work on the warehouses within the same EPT is enough.

domain_mmap first invokes *mmap* and then informs the kernel module about the domain mapping information. The kernel module records the information by using Linux's *rbtree* and validates domain page faults based on it. Huge page mapping is also supported through setting the *flag* argument. The last two interfaces are responsible for switching memory domains and are purely implemented in user mode except for accessing multiple domains in different EPTs. It is also necessary to know the current EPT-ID in user mode. For example, switching domains in the same EPT requires no VMFUNC. EPK does this by reusing the EPT-ID-Page During its initialization, the library asks the kernel module to map the EPT-ID-Page as read-only into the application. Besides, a domain memory allocator based on [34] is also provided.

Since servicing invocations from applications and recording the domain-related information, the kernel module provides a routine that aids in handling domain page faults. We insert a hook in the Linux page fault handler for invoking this routine. When a page fault occurs, the page fault handler still executes as before (e.g., allocates a free page) but invokes this routine just before setting the GPA of the newly allocated page in the page table entry. The routine then checks whether the page fault occurred within the domain regions and whether it was legal. If it is a legal domain page fault, the routine updates the GPA by adding *SHADOW REGION OFFSET* to it and invokes the hypercall to fill the mapping for the updated GPA in the EPT (as described in Section 3.1). Finally, the routine returns and the page fault handler sets the updated GPA in the page table entry. Another simple hook is added to the OS schedule function (i.e., *__schedule*). It saves/restores the EPT-ID for threads of applications that use EPK. Specifically, it saves the current EPT-ID in the thread's *task_struct* when scheduling out such a thread and restores the EPT-ID with *VMFUNC* (if necessary) when scheduling in the thread. Moreover, we add the VE handler for transparently supporting flexible multi-domain access.

In KVM, besides enabling VE and VMFUNC, we extend the hypercall handler to provide two additional functions for the guest kernel module. The first is to map the EPT-ID-Page, and the second is to add the EPT mapping for the VM's shadow address space. Furthermore, to support reclaiming the pages mapped in the shadow address space, the hypervisor needs to first disable VE on the pages to reclaim and record the reclaim information. When swapping back the pages, the hypervisor needs to re-enable VE on the pages. Besides these,

the hypervisor can reclaim the pages as before. Yet, this reclaiming mechanism is not supported in the current implementation of EPK.

EPK only requires minor modifications on Linux/KVM. Our prototype only adds 250 lines of code (LOC) in KVM, 13 LOC in guest OS, and 600 LOC in guest kernel module.

4 Case Study: Protecting Server Applications

Experiment Setup. All the experiments in this paper are conducted on a Dell PowerEdge R640 server with Intel Xeon Gold 6138 CPU. Hyper-threading is disabled, and the CPU frequency is fixed to 2.0GHz. The L2 TLB has 1536 entries. In Section 4 and Section 5, we implement and evaluate EPK on Linux/KVM-4.19.88 (both the guest OS and the hypervisor). The experiments are conducted in a VM (20 CPUs and 80GB memory), and the loopback network is used. All experiments use 4k memory pages without explicit statements.

Comparison Systems. Besides the native performance (run benchmarks with no isolation), we compare the performance of EPK with libmpk [38], lwC [32], and a *VMFUNC*-only solution. We evaluate libmpk in single-thread experiments since it does not support multi-threading. Since lwC is implemented on FreeBSD, we simulate its performance on Linux. Specifically, we first measure its switch cost (around 6,000 cycles, which corresponds to the reported data in Table 2 in ERIM [46] and Table 2 in lwC [32]) and then add such switch cost in the benchmarks (i.e., waiting for 6,000 cycles when switching context is needed). Note that the simulated performance will be better than the actual performance because the indirect cost of switching address space is ignored. We also implement a *VMFUNC*-only solution that provides one memory domain in one EPT and leverages *VMFUNC* for domain switches. The experiment of libmpk is conducted in host, while all the other systems run in the VM.

4.1 Micro-benchmarks

Domain Num	3	4	8	15	16	32	64
libmpk(128 pages)	184	184	186	188	12,991	13,148	13,048
VMFUNC	350	831	830	836	834	849	830
EPK	97	97	100	101	111	115	162

Table 4: The average cost (in cycles) of domain switches.

We leverage different solutions to create multiple memory domains and evaluate the domain switching cost (shown in Table 4). The test program initially runs in domain-0 (not counted in the domain number) and switches between the created domains in order (sequential access). The number of iterations is 100,000 and we measure the average cost. libmpk’s switch cost gets much higher when the domain number is above 15 (domain-0 takes one protection key). Besides,

its switch cost is severely influenced by the size of protected memory. When each domain contains 128 pages, its switch cost becomes more than 10,000 cycles if the domain number exceeds 16, which is even 100× slower than EPK. With the domain size increasing, its switch cost will enlarge due to more page table updates during key eviction, as shown in Table 2. In contrast, the switch time of the other two approaches is immune to the domain size.

The *VMFUNC*-only solution uses one EPT for domain-0, and its switch cost is about 350 cycles which mainly comes from two *VMFUNC* instructions when the domain number is no more than 3 (the total EPT number is no more than 4). However, its cost increases to around 830 cycles when the domain number exceeds 3. This is because TLB entries are tagged with EPT base addresses, and the involvement of more EPTs may decrease the TLB hit rate. Specifically, accessing the same memory page in different EPTs generates different TLB entries and then may exceed the capacity of the corresponding TLB set. EPK shows the lowest average switch cost since most switches are based on *WRPKRU*. When the domain number is less than 16, it outperforms libmpk because the latter one involves virtual protection key management (although no key eviction). When the domain number exceeds 60, the average cost of EPK increases since there are more than 4 EPTs.

Yet, in the *worst case*, EPK needs both *WRPKRU* and *VMFUNC* for switching to one domain, and takes around 860 cycles for traversing domains like the above, which means the performance of EPK may converge with the *VMFUNC*-only solution with random switches.

4.2 Macro-benchmarks

NGINX. Introducing intra-process memory isolation to server applications brings the potential to achieve higher security or reliability. We first apply different solutions to a widely-used web server, NGINX [9] v1.12.1, to evaluate the performance overhead. We isolate SSL session keys as ERIM [46] does (including preventing the abuse of domain switching), except that we store per-client session keys in different domains rather than in one domain. We leverage ab [1] to generate the workload: 300 clients keep sending file requests one by one. The server thread is fully loaded. The total domain number is 300 and each domain contains 5 memory pages.

Figure 4a shows the evaluation results. The throughput is normalized because libmpk is implemented on Linux 4.14.2 and the native throughput differs on Linux 4.14.2 and 4.19.88 (EPK) (while KPTI is disabled on both, other mitigations on CPU vulnerabilities are key factors). EPK imposes overhead from 4.3% to 5.8% compared with native and outperforms other solutions. The overhead of the *VMFUNC*-only solution varies from 11.0% to 12.4%. Notice that the NGINX serving thread handles client requests in order. Thus, most domain

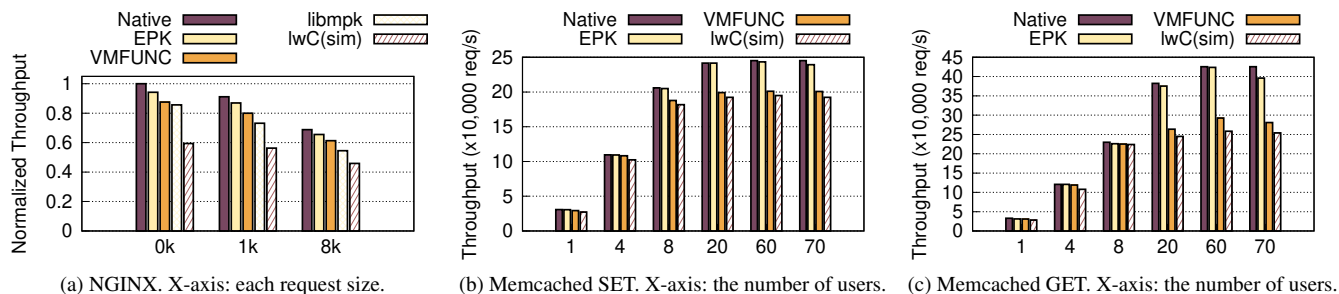


Figure 4: (a) shows the performance of protecting session keys in NGINX web server. (b) and (c) show the performance of isolating different users in Memcached (omit libmpk since it lacks the support of multi-threading).

switches in EPK need no EPT changing, making EPK outperform the *VMFUNC*-only solution. When storing each session key in an individual context in *lwC*, the overhead is 37.1% on average merely due to the explicit cost of domain switches. The overhead incurred by *libmpk* ranges from 14.5%-18.9% (23.4% to 33.2% if in the virtualization environment) due to the involvement of page table modifying and TLB flushing. In the cases of infrequent switching and small domain size (5 pages), *libmpk* will not lead to too much overhead.

Memcached. We evaluate Memcached [8] 1.6.9 and use *libMemcached* as the client library in this experiment. Memcached is a well-known key-value store and usually runs as a multi-thread server application. Arbiter [49] suggests that it is preferred to isolate data from different clients in Memcached for security-sensitive cases. Like Arbiter, we enable Simple Authentication and Security Layer (the SASL configuration) in Memcached and then isolate data stored by different clients. Besides, we slightly modify the request dispatching scheme of Memcached so that the requests from one client are always dispatched to the same worker thread for leveraging the domain affinity provided by EPK. The worker thread switches to the client’s corresponding domain before handling a request and exits that domain before sending back the reply. We create a different number of client threads, and each of them uses *libMemcached* for sending SET/GET requests. The sizes of key and value are 32 bytes and 256 bytes, separately. There are four worker threads (default configuration) on the server-side, and the clients will be evenly partitioned to them. In this experiment, the max domain number is 70 and each domain contains about 2,000 memory pages.

Figure 4b and 4c show the throughput of Memcached. As before, *lwC* leads to the highest overhead due to its expensive switch cost. When the client number is no more than 60, EPK incurs at most 0.7% overhead on the throughput of SET operations. The overhead on the throughput of GET operations is slightly higher (up to 2.9%) because the GET operations are lighter than the SET ones. The extremely low overhead is because no EPT switches happen on the critical path. EPK allows each worker thread to create 15 domains in one EPT, and thus four worker threads can handle 60 clients (60 domains)

without switching EPTs. When the number of clients exceeds 60, the overhead of EPK becomes larger because some worker threads need to handle requests from more than 15 clients, and then EPT switches happen.

In contrast, the *VMFUNC*-only solution incurs a much larger overhead, i.e., up to 17.9% and 34.0% overhead for the throughput of GET and SET operations, separately. The overhead mainly comes from TLB misses, as explained in Section 4.1. For validation, we further carry out two experiments marked as *VMFUNC-test-1* and *VMFUNC-test-2* in Table 5. The former one is that the worker thread switches to the target EPT and immediately switches back before handling a request. So, all the requests are handled in EPT-0. The latter one is that each worker thread always switches to EPT-1 for handling requests. So, all the requests are handled in EPT-1. Both of them show close-to-native performance, and the TLB miss number is not significantly enlarged. However, the *VMFUNC*-only solution causes many more TLB misses and then leads to the highest overhead. The overhead of TLB miss in NGINX is not obvious because the worker thread of NGINX only switches to other EPTs when accessing session keys while the worker thread of Memcached executes most logic in different EPTs.

	Throughput (×10K req/s)	dTLB/TLB misses
<i>Native</i>	24.5	1 / 1
<i>VMFUNC-test-1</i>	24.4	1.1 / 2.4
<i>VMFUNC-test-2</i>	24.0	1.2 / 2.4
<i>VMFUNC</i>	20.1	9.5 / 29.1

Table 5: The throughput and TLB misses (normalized) when evaluating Memcached SET operation with 60 clients.

Since *libmpk* does not support multi-thread, we evaluate *libmpk* in Memcached with a single worker thread. When there are 60 domains, the overhead of the above test exceeds 80%, which is significantly higher than that in NGINX because each domain contains about 2,000 pages.

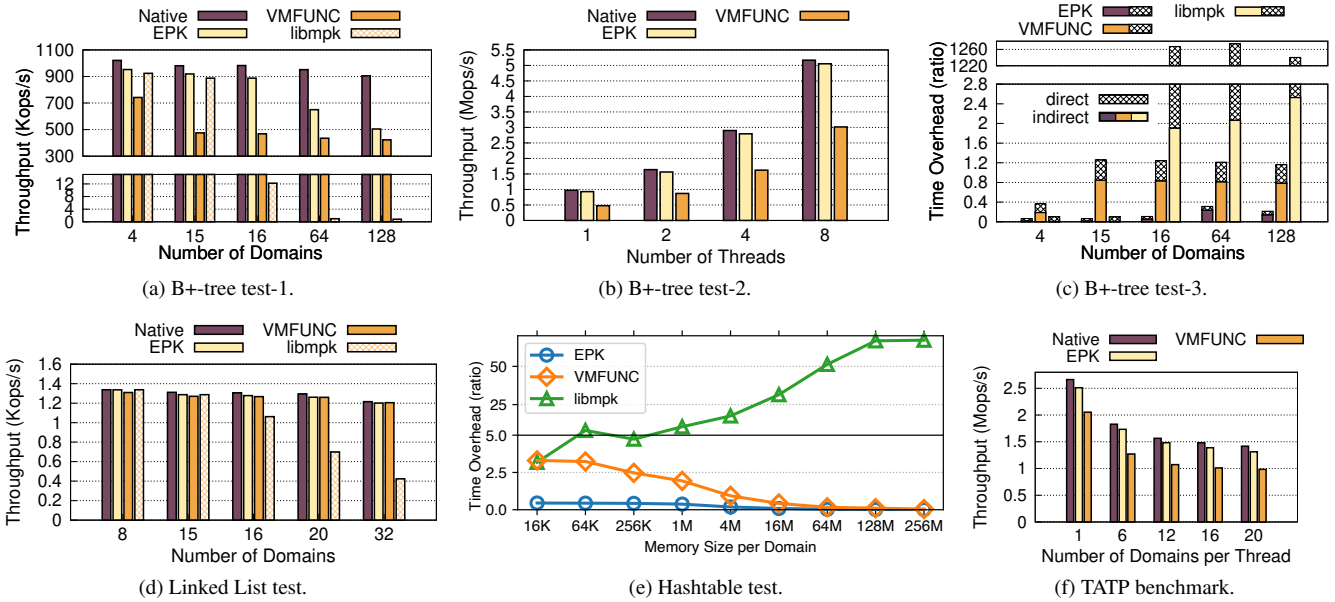


Figure 5: B+-tree (a) Single-thread and random access, (b) Multi-thread and each thread operates on 15 domains, (c) Time breakdown when single-thread and sequential access. (d) Linked List (low switch frequency). (e) Hashtable benchmark (different domain memory size). (f) TATP benchmark. Omit libmpk (single-thread support) in the multi-threaded cases.

5 Case Study: Isolating NVM Data

To embrace the low access latency of NVM, applications usually map NVM into the address space and access it through load/store instructions. Bringing intra-process memory isolation to protect NVM data (e.g., reducing the data exposure time) has also been investigated by recent work [56, 57]. In this section, we evaluate the benchmarks similar to existing NVM studies [23, 33, 56, 57], using DRAM as NVM.

5.1 Data Structure Benchmarks

We first experiment on B+-tree. We map each B+-tree in an individual domain and create different numbers of threads to do lookup or insert operations (the ratio is 1:1 and other ratios show similar performance trends). Domain switches occur before and after an operation. Each tree initially has 500,000 key-value pairs, and each tree node has up to 32 child nodes. In this experiment, the max domain number is 128 and the size of each domain is about 128MB.

The overhead of lwc in the above benchmarks is always around 80% because one B+-tree operation takes just about 2,100 cycles.

Figure 5a shows the throughput when a single thread operates on a randomly selected tree (i.e., randomly switching to a domain). If the domain number is less than 16, EPK and libmpk bring about 7% and 11% overhead, individually. When the domain number exceeds 15, libmpk introduces unacceptable overhead (throughput drops by 99.8%) due to the substantial cost of key eviction (as the domain size is not small), and EPK can outperform it by two orders of magni-

tude. Note that the kernel version has minor effects on the native performance since this benchmark rarely issues system calls. The *VMFUNC*-only solution incurs 27% overhead when the domain number is 4. Compared to EPK, its higher overhead comes from two sources: one is *VMFUNC* is slower than *WRPKRU*; the other is more TLB misses (its dTLB and iTLB misses are 1.34× and 3.34× of EPK’s, respectively). When the domain number increases to 64 and 128, EPK’s overhead also increases to 32% and 44% because more EPTs and EPT switches are required. Specifically, when there are 64 domains, 78% of domain switches in EPK involve EPT switches. If accessing different domains sequentially instead of randomly, EPK’s overhead is below 10% (3% for huge page) when the domain number is no over 60.

Figure 5b shows the performance when there are multiple threads on different cores and each thread accesses 15 different domains. EPK’s overhead remains below 5% as the thread number increases, which is significantly lower than that of the *VMFUNC*-only solution (41% to 51%).

We further analyze the overhead of the three approaches in terms of the time cost (the part that exceeds the native time): the switching time (direct cost) and the rest time overhead incurred by the pollution on CPU internal structures including TLBs and caches (indirect cost). The experiment is one thread operates on the tree in each domain sequentially to complete a fixed amount of operations. Figure 5c presents the breakdown. The *VMFUNC*-only solution brings about 1.2× time overhead when the domain number exceeds 8. Its indirect cost remains around 0.8× because the TLB miss rate is almost stable. The page table updating operations of libmpk leads to both high direct cost and indirect cost (not only incurs TLB misses

but also leads to intensive cacheline pollution). EPK causes $0.23 \times$ ($0.13 \times$ for huge page) indirect cost when there are 64 domains due to more than 4 EPTs, which is still better than others, and causes much lower cost for fewer domains.

Note that the domain switch frequency is proportional to the throughput in the presented benchmarks. We also conduct an experiment on Linked List (Figure 5d): each list is separated into one domain, and one thread performs 10 operations (search, insert, delete) in a random list for each time. The switch frequency is less than 1,400 times per second. libmpk still introduces 65.1% performance overhead when there are 32 domains and each domain is 256 MB. The other two approaches cause unnoticeable overhead.

Last, Figure 5e shows the overhead (in terms of the time cost) of different approaches as the domain size increases. In this experiment, each hash table resides in one domain (32 domains), and one thread keeps performing an operation (search or insert) in one random domain. We gradually increase the domain size by adding more buckets/key-value pairs in each hash table. The overhead of libmpk increases as the domain size grows, as expected, whereas the overhead of EPK and the VMFUNC-only solution decreases because the native performance decreases when more memory involves. Specifically, when each domain is 256 MB, the overhead of the latter two are 1.3% and 4.7%, respectively.

Virtualization Cost. Virtualization brings performance overhead to applications, especially when the working set is large and TLB misses are frequent. For example, when the domain size in hash table is configured as 16KB or 128MB, the virtualization overheads are 2.1% and 9.0%, respectively. When a VM application uses EPK, the virtualization cost is not accounted on EPK. Otherwise (in bare metal), the virtualization cost should be included in the overhead of EPK. Nevertheless, a thin virtualization layer instead of a full-fledged hypervisor can minimize the virtualization cost [22].

5.2 OLTP Benchmarks

TATP [43] is an online transaction processing (OLTP) benchmark. In the experiment, we use the above B+-tree as the data store and create four threads to execute transactions (three read-only and three read-write ones). We store a fixed amount of initial data in different domains, and each thread switches to the corresponding domain before executing one transaction. The max domain number is 80 and the size of each domain is 512MB. Figure 5f presents the throughput as the domain number for each thread increases from 1 to 20. The native throughput is in a decreasing trend along with the increase of domain number because more data weakens the cache locality. EPK's overhead is within 7%, while the VMFUNC-only solution incurs up to 32% overhead. We also run single-thread TATP with libmpk. Similar to B+-tree test-1, the overhead of libmpk is over 99% when the domain number exceeds 15.

TPC-C [18] is another OLTP benchmark in which there

are multiple warehouses. We isolate different warehouses as well as their associated data in different domains. The max domain number is 128 and the size of each domain is 400MB. According to its specification, 7.2% of the transactions update multiple warehouses simultaneously. There are also four threads executing the transactions. When each thread operates on less than 16 domains, EPK achieves almost the same throughput as the native (0.6% overhead). The overhead is lower than that in TATP because the transactions in TPC-C are more heavyweight. When each thread operates on 32 domains, the overhead of EPK becomes 3.2% as VEs are triggered for supporting transparent multi-domain access. The other approaches are infeasible in this experiment due to the lack of the support of multi-domain access.

6 Case Study: Boosting IPCs in Microkernels

6.1 HyBridge

Different from monolithic OSes which run all the OS modules in the kernel-level, microkernels leave minimal functionalities in the kernel while running all other OS modules (referred to as system servers below) such as file systems, network stacks, and device drivers into separated user-level processes. Inherently, microkernels embrace better security and fault isolation, but leads to non-negligible communication cost at runtime. Specifically, since system servers are user-level processes, the interactions between two servers or between an application and a server require inter-process communication (IPC). In contrast, on monolithic OSes (e.g., Linux), the interaction between two OS modules only requires function calls, and the interaction between applications and the OS can be as fast as about 150 cycles (*syscall* and *sysret*). So there has been a long line of research to reduce the cost of IPC to bridge the performance gap between microkernels and monolithic OSes.

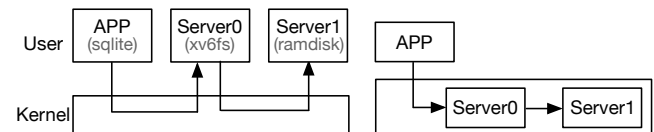


Figure 6: Traditional IPC flow on microkernels is shown on the left, and IPC with UnderBridge is shown on the right.

A most recent IPC design called UnderBridge [22] retrofits Intel MPK to optimize (synchronous) IPC. For reducing the cost of IPC between an application and a server, it pulls system servers from user-level processes into the kernel address space as shown in Figure 6. Besides, it leverages Intel MPK to ensure the isolation between system servers in the kernel, and the IPCs between them are based on *WRPKRU* and thus greatly optimized. However, due to the limitation of MPK memory domains, it can only run limited system servers in the kernel and accelerate IPCs to them (issue-1). Also, although it can reduce the privilege switches during IPCs between ap-

applications and servers, the page table switches are still needed because it requires a separate kernel page table (issue-2).

Since EPK can construct even thousands of isolated memory domains efficiently and enable fast domain switch at user-level, we propose EPK-based HyBridge for boosting IPCs for microkernels, which is inspired by UnderBridge while fixing the two issues of UnderBridge. As shown in Figure 7, system servers run at user-level, and each one exclusively takes one or more memory domains for holding its own memory, including code, data, stack, and heap. Thus, one system server cannot access others' private memory, just like when they are isolated in different processes while IPCs are based on domain switches.

Cross-server IPC. The cross-server IPCs only happen between system servers that need to interact with each other. For example, a file system communicates with a disk driver while a network stack does not. This also matches the *domain affinity* in EPK. Therefore, the microkernel can run the related system servers in the same EPT. When two servers establish an IPC connection, the microkernel will map an IPC gate, i.e., a piece of code, for them. During an IPC invocation, the gate will transfer the control flow from the caller to the callee. Specifically, it saves the caller's execution states, then executes *WRPKRU* to switch to the callee's domain, and restores the callee's execution states. Similarly, it does the reverse procedure when the IPC returns. HyBridge also allows two servers to share memory for exchanging data by assigning a free memory domain to them, e.g., shared memory domain 4 in Figure 7.

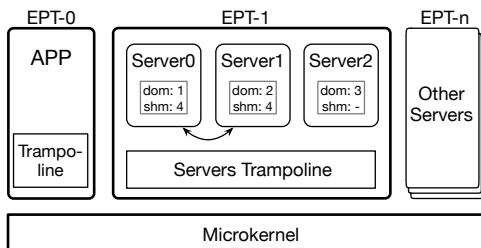


Figure 7: The overview of HyBridge. The numbers after colons are domain IDs. Shared memory is short as shm.

Application-to-server IPC. Applications execute in different processes (in EPT-0) just like before while several system servers can run in one process (across one or more EPTs), which means each application has a unique guest CR3 (GPA) while multiple servers share one. Since an application and a server run in different EPTs, the IPCs between them need EPT switching. HyBridge attaches a trampoline in the EPTs for running servers and maps the trampoline into an application when it asks for establishing an IPC connection with some server. The trampoline plays the role of the IPC gate and uses *VMFUNC* to switch between the caller and the callee. Though *VMFUNC* can directly switch EPT, it does not change guest CR3. However, for an application-to-server IPC, the caller and callee use different CR3 (CR3-App and CR3-Server). So,

besides mapping the trampoline, HyBridge also maps CR3-App (GPA) to the HPA of CR3-Server in the server's EPT during the IPC establishment. In this way, the HPA mapping for the guest CR3 is transparently changed after executing *VMFUNC*, i.e., the guest page table is switched from the application to the server. When an application invokes an IPC, the trampoline saves the caller's execution states (executes in EPT-0), executes *VMFUNC* (switches the EPT), and restores the callee's execution states (executes in server's EPT).

Security Enforcement. Besides memory isolation, HyBridge employs additional security mechanisms to achieve the same security guarantee as original microkernels. Compared with original IPC designs, HyBridge makes an untrusted system server have two more potential attack vectors. One is that a server may bypass the memory isolation by maliciously executing *WRPKRU* or *VMFUNC* and then access others' memory. The other is that a server may issue arbitrary IPCs to other servers by maliciously executing the trampoline code without the corresponding capabilities.

HyBridge eliminates the two attack vectors as follows. First, it utilizes binary scanning and rewriting to ensure that each server contains no *WRPKRU* or *VMFUNC* instructions during binary loading. Meanwhile, it adds sanity checks in the IPC gates for ensuring the argument of *WRPKRU* is legal, which is similar as ERIM [46]. So, a compromised or malicious server cannot illegally execute these two instructions to retrieve unauthorized memory permissions even with return-oriented programming (ROP). Second, HyBridge uses a token-based mechanism to authenticate IPC invocations as SkyBridge [37] does. Considering control flow hijacking, trampolines can be executed arbitrarily or it is even possible to jump into the middle of the trampoline, i.e., using *VMFUNC* to switch to any EPT. Although they cannot be misused to break the memory isolation, an untrusted server may issue arbitrary IPCs by invoking them. To prevent this, HyBridge lets a server generate a random 64-bit token for a registered client (another server or an application) when building the IPC connection, and a client needs to pass the token during IPCs for authentication. The server only serves the IPC requests with legal tokens, so the problem of arbitrary IPCs can be avoided. Moreover, HyBridge also prevents the occurrence of *VMFUNC* in applications by scanning and rewriting the binary code. Thus, an application can only switch to system servers through the mapped trampoline.

6.2 Experiments

We implement HyBridge on three well-known microkernels, Zircon [4], seL4 [10], and Fiasco.OC [3], to assess its effectiveness. Besides, we also compare it with SkyBridge [37] which runs system servers in different EPTs and implements kernel-bypass IPCs based on *VMFUNC*, and UnderBridge [22]. We deploy the thin virtualization layer from SkyBridge while applying extensions needed by three IPC de-

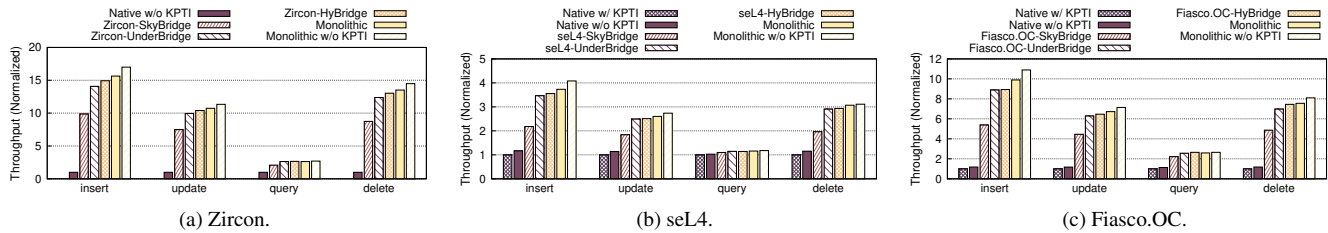


Figure 8: Normalized throughput of SQLite3 on different microkernels. KPTI is short for kernel-page-table-isolation.

signs. We evaluate the performance of SQLite3 v3.23.0 [11], a database application, after applying different IPC mechanisms on different microkernels. For severing SQLite3, we run two system servers, a file system named xv6fs [16] and a RAMdisk. When SQLite3 operates a storage file, it will first invoke the xv6fs server by an application-to-server IPC; then, the xv6fs will access RAMdisk through cross-server IPCs. We also simulate the performance of a monolithic kernel by running system servers in the kernel and connecting them with function calls.

Figure 8a, 8b and 8c present the normalized throughput on the three microkernels. The native performance of each microkernel is set as the baseline. Because Zircon has the slowest native IPC among the three microkernels since it includes scheduling overhead in IPCs, HyBridge can provide the highest speedup for it, i.e., more than $9\times$ speedup for three database operations. The performance improvement of query operations is relatively small because SQLite3 has an internal cache of recent data and may handle the queries without issuing IPC requests. For seL4 which optimizes IPC performance extensively, HyBridge can also improve the throughput (except query) to more than $2.5\times$ of the native.

Besides, HyBridge can outperform SkyBridge by up to 66% because most IPCs issued from SQLite3 to xv6fs involve multiple cross-server IPCs between xv6fs and RAMdisk, whereas the cross-server IPCs are more lightweight in HyBridge. Specifically, a cross-server IPC takes 110 and 437 CPU cycles in HyBridge (*WRPKRU*-based) and SkyBridge (*VMFUNC*-based), respectively. In this benchmark, HyBridge only shows slightly higher performance than UnderBridge since cross-server IPCs dominate, while it has more advantage over UnderBridge in the application-to-server IPC (e.g., 527 vs. 723 CPU cycles when implemented on our research microkernel, ChCore [22]) owing to no CR3 changing.

7 Other Related Work

Many studies [15, 19, 27, 36, 42, 48, 58] leverage instruction instrumenting to achieve memory isolation, which may incur non-trivial overhead. Many other studies [25, 27, 32, 35, 39] utilize the memory management unit (MMU) to check memory accesses efficiently. Specifically, they divide a process into different compartments and assign each one an individual (extended) page table. However, switching between compart-

ments requires (extended) page table switching, which can be costly when the cross-boundary invocation is frequent. Twizzler [14] is a pioneer data-centric OS for NVM and uses EPT/VMFUNC to create different memory domains for NVM isolation. Differently, EPK focuses on solving the challenges of combining MPK and EPT/VMFUNC and outperforms a VMFUNC-only solution. Besides, recent work [28, 51] harnesses hardware features like Supervisor-Mode Access Prevention (SMAP) or underused intermediate privilege levels (Ring1 and Ring2 on x86) to achieve efficient intra-process memory isolation. Yet, they can only provide two isolated memory domains.

Prior work [21, 47, 52, 54] also proposes architecture designs to facilitate efficient intra-process memory isolation, which, however, is not achievable on commodity machines. PLB [53] proposes architecture changes which differs from Intel MPK for supporting scalable domains but requires virtually indexed cache which may cause performance issues. Besides Intel, both ARM (ARMv7) and AMD propose similar features of memory domains and face the same scalability problem of the domain number. The basic idea of EPK is feasible to be extended to them as they also support 2-stage address translation. Yet, for efficiency, hardware-assisted fast switching (currently commercially unavailable) of stage-2 page table is needed on the two architectures.

An orthogonal study [17] shows that some system calls can be used to break the MPK isolation, so the OS may need to be aware of MPK in the future or the applications needs to incorporate other mechanisms like system call filtering [12].

8 Summary

This paper presents EPK which first combines the usage of MPK and hardware virtualization features to achieve scalable and efficient intra-process memory isolation. The case studies demonstrate various potential usages of EPK.

9 Acknowledgement

We sincerely thank the anonymous shepherd and reviewers for their insightful suggestions. This work is supported in part by China National Natural Science Foundation (No. 61925206 and No.U19A2060), Huawei, and STCSM (No. 21511101502). Yubin Xia is the corresponding author.

References

- [1] <https://httpd.apache.org/docs/2.4/programs/ab.html>.
- [2] The 20 best linux apps ever. <https://helpdeskgeek.com/linux-tips/the-20-best-linux-apps-ever/>.
- [3] Fiasco.oc repository. <https://l4re.org/download/snapshots/>.
- [4] Fuchsia repository. https://fuchsia.dev/fuchsia-src/development/source_code.
- [5] The heartbleed bug. <https://heartbleed.com/>.
- [6] Intel optane technology. <https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html>.
- [7] Intel software developer's manual. <https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf>.
- [8] Memcached. <https://www.memcached.org>.
- [9] Nginx. <https://nginx.org>.
- [10] sel4 repository. <https://github.com/seL4/seL4>.
- [11] Sqlite. <https://www.sqlite.org/index.html>.
- [12] Jenny: Securing syscalls for PKU-based memory isolation systems. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, Aug. 2022. USENIX Association.
- [13] A. Ahmad, S. Lee, P. Fonseca, and B. Lee. Kard: Lightweight data race detection with per-thread memory protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 647–660, New York, NY, USA, 2021. Association for Computing Machinery.
- [14] D. Bittman, P. Alvaro, P. Mehra, D. D. E. Long, and E. L. Miller. Twizzler: a data-centric os for non-volatile memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 65–80. USENIX Association, July 2020.
- [15] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 45–58, New York, NY, USA, 2009. Association for Computing Machinery.
- [16] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery.
- [17] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard. Pku pitfalls: Attacks on pku-based memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1409–1426. USENIX Association, Aug. 2020.
- [18] T. P. P. Council. <http://www.tpc.org/tpcc/>. *TPC Benchmark C*.
- [19] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.
- [20] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 137–151, New York, NY, USA, 1996. ACM.
- [21] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi. IMIX: In-process memory isolation extension. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 83–97, Baltimore, MD, Aug. 2018. USENIX Association.
- [22] J. Gu, X. Wu, W. Li, N. Liu, Z. Mi, Y. Xia, and H. Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 401–417, July 2020.
- [23] J. Gu, Q. Yu, X. Wang, Z. Wang, B. Zang, H. Guan, and H. Chen. Pisces: A scalable and efficient persistent transactional memory. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 913–928, USA, 2019. USENIX Association.
- [24] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 489–503, Berkeley, CA, USA, 2019. USENIX Association.
- [25] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 393–405, New York, NY, USA, 2016. Association for Computing Machinery.
- [26] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.
- [27] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanopoulos. No need to hide: Protecting safe regions on

- commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys '17*, pages 437–452, New York, NY, USA, 2017. ACM.
- [28] H. Lee, C. Song, and B. B. Kang. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, pages 1441–1454, New York, NY, USA, 2018. ACM.
- [29] H. Lefeuvre, V.-A. Bădoiu, c. Teodorescu, P. Olivier, T. Mosnoi, R. Deaconescu, F. Huici, and C. Raiciu. Flexos: Making os isolation flexible. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS '21*, page 79–87, New York, NY, USA, 2021. Association for Computing Machinery.
- [30] J. Liedtke. Improving ipc by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, SOSP '93*, pages 175–188, New York, NY, USA, 1993. ACM.
- [31] J. Liedtke. A persistent system in real use - experiences of the first 13 years. pages 2 – 11, 01 1994.
- [32] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. Light-weight contexts: An os abstraction for safety and performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16*, pages 49–64, Berkeley, CA, USA, 2016. USENIX Association.
- [33] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17*, page 329–343, New York, NY, USA, 2017. Association for Computing Machinery.
- [34] R. Liu and H. Chen. Ssmalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the Asia-Pacific Workshop on Systems*, pages 1–6, 2012.
- [35] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15*, pages 1607–1619, New York, NY, USA, 2015. ACM.
- [36] S. McCamant and G. Morrisett. Evaluating sfi for a cisc architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.
- [37] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019, EuroSys '19*, pages 9:1–9:15, New York, NY, USA, 2019. ACM.
- [38] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. Libmpk: Software abstraction for intel memory protection keys (intel mpk). In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '19*, pages 241–254, Berkeley, CA, USA, 2019. USENIX Association.
- [39] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis. xmp: Selective memory protection for kernel and user space. In *Proceedings of 41st IEEE Symposium on Security and Privacy, S&P '20*, 2020.
- [40] V. A. Sartakov, L. Vilanova, and P. Pietzuch. Cubicleos: A library os with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021*, page 546–558, New York, NY, USA, 2021. Association for Computing Machinery.
- [41] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss. Donky: Domain keys – efficient in-process isolation for risc-v and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, Aug. 2020.
- [42] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX Conference on Security, USENIX Security'10*, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.
- [43] N. Simo, W. Antoni, m. Markku, and R. Vilho. <http://tatpbenchmark.sourceforge.net/>. *Telecom Application Transaction Processing Benchmark*.
- [44] M. Sung, P. Olivier, S. Lankes, and B. Ravindran. Intra-ukernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 143–156, 2020.
- [45] D. Tsafirir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Experimental Computer Science on Experimental Computer Science, ecs'07*, pages 3–3, Berkeley, CA, USA, 2007. USENIX Association.
- [46] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. Erim: Secure, efficient in-process isolation with protection keys (mpk). In *Proceedings of the 28th USENIX Conference on Security Symposium, SEC'19*, pages 1221–1238, Berkeley, CA, USA, 2019. USENIX Association.
- [47] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. Codoms: Protecting software with code-centric memory domains. In *2014 ACM/IEEE 41st Inter-*

- national Symposium on Computer Architecture (ISCA)*, pages 469–480. IEEE, 2014.
- [48] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, SOSP '93*, pages 203–216, New York, NY, USA, 1993. ACM.
- [49] J. Wang, X. Xiong, and P. Liu. Between mutual trust and mutual distrust: Practical fine-grained privilege separation in multithreaded applications. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '15*, page 361–373, USA, 2015. USENIX Association.
- [50] X. Wang, S. Yeoh, P. Olivier, and B. Ravindran. Secure and efficient in-process monitor (and library) protection with intel mpk. In *Proceedings of the 13th European Workshop on Systems Security, EuroSec '20*, page 7–12, New York, NY, USA, 2020. Association for Computing Machinery.
- [51] Z. Wang, C. Wu, M. Xie, Y. Zhang, K. Lu, X. Zhang, Y. Lai, Y. Kang, and M. Yang. Seimi: Efficient and secure smap-enabled intra-process memory isolation. *ieee symposium on security and privacy*, 2020.
- [52] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. Fast protection-domain crossing in the cheri capability-system architecture. *IEEE Micro*, 36(5):38–49, Sept. 2016.
- [53] J. Wilkes and B. Sears. A comparison of protection lookaside buffers and the PA-RISC protection architecture. In *Technical Report HPL-92-55*. Hewlett-Packard Laboratories, Mar. 1992.
- [54] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS X*, pages 304–316, New York, NY, USA, 2002. ACM.
- [55] M. Wu, Z. Zhao, Y. Yang, H. Li, H. Chen, B. Zang, H. Guan, S. Li, C. Lu, and T. Zhang. Platinum: A cpu-efficient concurrent garbage collector for tail-reduction of interactive services. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 159–172. USENIX Association, July 2020.
- [56] Y. Xu, Y. Solihin, and X. Shen. Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20*, page 987–1000, New York, NY, USA, 2020. Association for Computing Machinery.
- [57] Y. Xu, C. Ye, Y. Solihin, and X. Shen. Hardware-based domain virtualization for intra-process isolation of persistent memory objects. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 680–692. IEEE, 2020.
- [58] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy, SP '09*, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.



Memory Harvesting in Multi-GPU Systems with Hierarchical Unified Virtual Memory

Sangjin Choi*
KAIST

Taeksoo Kim*
KAIST

Jinwoo Jeong
Ajou University

Rachata Ausavarungnirun
KMUTNB

Myeongjae Jeon
UNIST

Youngjin Kwon
KAIST

Jeongseob Ahn[†]
Ajou University

Abstract

With the ever-growing demands for GPUs, most organizations allow users to share the multi-GPU servers. However, we observe that the memory space across GPUs is not effectively utilized enough when consolidating various workloads that exhibit highly varying resource demands. This is because the current memory management techniques were designed solely for individual GPUs rather than shared multi-GPU environments.

This study introduces a novel approach to provide an illusion of virtual memory space for GPUs, called hierarchical unified virtual memory (HUVVM), by incorporating the temporarily idle memory of neighbor GPUs. Since modern GPUs are connected to each other through a fast interconnect, it provides lower access latency to neighbor GPU's memory compared to the host memory via PCIe. On top of HUVVM, we design a new memory manager, called memHarvester, to effectively and efficiently harvest the temporarily available neighbor GPUs' memory. For diverse consolidation scenarios with DNN training and graph analytics workloads, our experimental result shows up to $2.71 \times$ performance improvement compared to the prior approach in multi-GPU environments.

1 Introduction

As the demand for GPUs explodes, it is now a common practice in both academia and industry to equip multiple GPUs in a single server and make them shareable. Many enterprises in the industry have built large GPU clusters comprised of a set of multi-GPU servers to satisfy the demand for a variety of workloads from deep learning [1, 13, 18, 26, 36] to graph applications [6, 10, 19, 31] while saving the infrastructure cost by sharing. However, as a downside, achieving high GPU resource efficiency in such multi-GPU servers remains a challenge. Figure 1 presents that the current memory management technique is not effective enough for shared multi-GPU

environments where multiple jobs are running across GPUs independently. Although a small amount of memory ranging from hundreds of MB to a few GB remains idle in one or a few GPUs, other GPUs under heavy memory pressure rely on the host memory as a swap device that is significantly slower than remote GPUs within the same server.

Meanwhile, GPU vendors have faced the challenge of scaling the memory capacity of single GPUs. To overcome the limited capacity of GPUs, a train of previous studies provides an illusion of infinite memory space with the host memory [11, 14, 17, 25, 28]. However, none of the work does utilize the idle memory of neighbor GPUs in commodity multi-GPU systems. As modern GPU servers are commonly equipped with 8~16 GPUs connected via high-speed interconnect such as NVLink, accessing the idle memory of neighbor GPUs is much faster than that of the host. For instance, NVIDIA DGX-2 has 16 GPUs with point-to-point connections through NVLink 3.0, yielding a large pool of 512GB GPU memory at 600GB/s bidirectional bandwidth [23]. On the other hand, swapping GPU memory to host DRAM via the latest PCIe 4.0 could utilize up to 32GB/s bandwidth only.

In this study, we introduce a new approach providing an illusion of virtual memory space for GPUs called *hierarchical unified virtual memory (HUVVM)* comprised of local GPU, spare memory of neighbor GPUs, and the host memory. HUVVM opens up a new opportunity for memory virtualization by increasing the effective memory space with minimal performance overhead. When the local GPU memory does not have free space, HUVVM leverages the spare¹ memory in neighbor GPUs *as a victim cache* between the GPU and host instead of directly swapping out data to the host memory.

However, it is challenging to effectively and efficiently harvest the spotty-available, small fraction of neighbor GPUs' memory because the amount of idle memory is highly variable and unknown a priori. Beyond the single GPU perspective, we redesign the memory management scheme for modern multi-GPU servers. HUVVM systems have to find the best

*Co-first authors

[†]Corresponding author

¹The terms *spare*, *idle*, and *harvested* are used interchangeably.

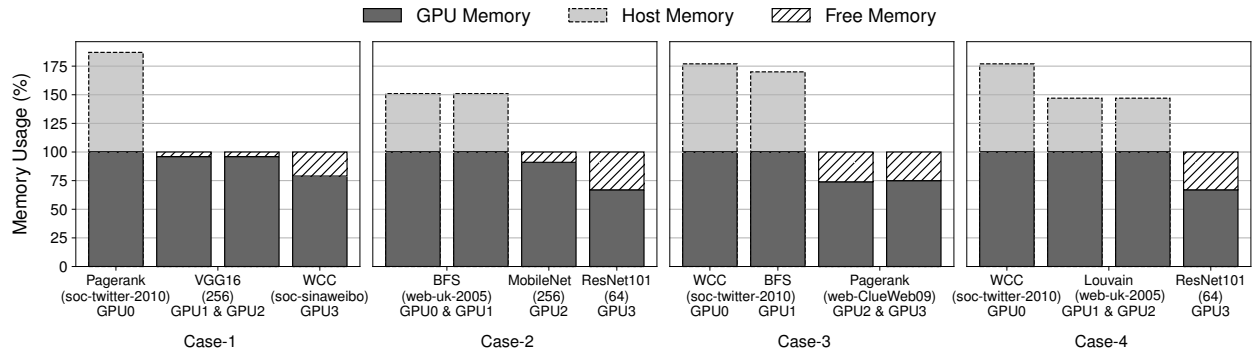


Figure 1: Memory usage snapshot in multi-GPUs hosting memory-intensive workloads (Section 6.1 and Table 2 present the detailed information for workloads and experimental environment)

way to utilize the small fraction of harvested memory while minimizing the performance impact on workloads running in the neighbor GPUs.

To that end, we propose a memory manager of HUVm, called *memHarvester*, implemented in the GPU driver layer. *memHarvester* functions as a centralized coordinator for data-path in HUVm. As an essential part of *memHarvester*, we propose a new multi-path parallel prefetcher, which exploits the path diversity of HUVm, comprised of PCIe and NVLink. Unlike many previous approaches [11, 14, 17, 25, 28] relying only on the host memory via PCIe, *memHarvester* first prefetches data from the spare memory to the local GPU through NVLink. Meanwhile, if the PCIe channel attached to the neighbor GPU is not contended, *memHarvester* allows for prefetching the data from the host memory to the spare memory of the neighbor GPU through the PCIe channel in parallel. Therefore, we can convert the latency of fetching data from the host memory to that of the spare memory effectively. *memHarvester* manages the space of harvested memory. Due to the limited space of spare memory, *memHarvester* is unable to keep all the evicted data in the harvested memory, leading to the host memory swap eventually. To reduce the performance overhead of data eviction from GPU to host, *memHarvester* supports eviction with 2MB large pages to host memory instead of 4KB base pages.

When HUVm and *memHarvester* host multiple workloads, it can improve memory utilization and overall server efficiency. However, on a downside, *memHarvester* may cause performance interference to the applications running on the GPU yielding idle memory. Thus, *memHarvester* immediately reclaims the spare memory to give it back to its original physical memory space with minimal latency whenever the application running on the yielding GPU needs additional memory, thereby minimizing performance interference.

We implement our prototype system on top of NVIDIA’s unified virtual memory (UVM) driver version 460.67, which is publicly accessible [29]. Without any modifications to applications or machine learning platforms, *memHarvester* transpar-

ently detects the availability of spare memory and dynamically constructs a new memory hierarchy. We quantify the effectiveness of *memHarvester* with HUVm for diverse consolidation scenarios on an AWS p3.8xlarge instance. The server has four NVIDIA V100 (16GB) GPUs connected through NVLink. Our experimental result shows that *memHarvester* can achieve significant throughput improvement for the large graph analytics workloads. For diverse consolidation scenarios with DNN training and graph analytics workloads, our experimental result shows up to $2.71\times$ performance improvement compared to the prior approach in multi-GPU environments with minimum interference of other workloads running on the same server.

2 Motivation and Background

This section characterizes memory usage behaviors of emerging workloads in shared multi-GPU environments and discusses opportunities to improve overall memory utilization by exploiting idle memory of neighbor GPUs.

2.1 Memory Usage in Shared Multi-GPUs

With the ever-growing demands of GPUs from development to deployment, most organizations allow semi-trusted users (e.g., employees in a company) to share the multi-GPU servers, reducing the cost of building the infrastructure [9, 13, 18, 35, 36]. Due to the shared nature, such GPU servers consolidate a wide range of workloads. In particular, many of the jobs running in the shared GPU servers are DNN training workloads for computer vision and natural language processing [11, 25, 28] that take a long time to complete. Thus, it leads to limited resource availability of certain GPUs at a time. Another widely witnessed application in multi-GPU servers is graph analytics [10, 19, 31], which figures out the relationships between objects in a given graph.

When looking at the memory consumption of such two emerging workloads, it is usually required for DNN training

jobs to tune the batch size to almost fit on GPUs to achieve maximum resource utilization [28]. On the other hand, the memory consumption of graph analytics jobs depends on the number of edges and vertices of a given graph. For a large dataset that does not fit in the given GPU memory, one can leverage a graph partitioning approach to make each partition fit on individual GPUs. However, the graph partitioning task introduces additional complexity in implementation [4] such that some of the graph algorithms are not supported to run on multi-GPUs (e.g., *Betweenness Centrality* in *cu-Graph* [6]). To overcome the memory space limitation, we can leverage host-side memory as a swap device to the GPUs through the unified virtual memory (UVM) technique that provides an illusion of infinite memory space [29]. Although this enables us to run analytics on large graphs or DNN training with large batches without out-of-memory errors, it significantly degrades performance.

In shared multi-GPU servers, we observe that a small amount of memory space across GPUs remains idle. Figure 1 shows memory snapshots of a 4-GPU (V100) server hosting multiple memory-intensive workloads. We profile four running scenarios each of which runs either graph analytics or DNN training jobs using one or multiple GPUs. The experimental environment is described in Section 6.1. The result shows that some GPU memory is not fully utilized, causing memory imbalance across workloads. In *Case-1*, *VGG16* and *WCC* leave a small amount of memory of GPU-1, 2, and 3, whereas *Pagerank* has to use the host memory for a swap device to run the large dataset. Another example is to run *Pagerank* with a relatively small dataset (*web-ClueWeb09*) with two GPUs in *Case-3*. In this case, the memory of GPU-2 and 3 is not fully utilized. Even though *Louvain* and *BFS* experience heavy memory pressure, the current memory management technique does not leverage the idle memory of the neighbor GPUs. These results confirm that the current memory management design is still not efficient in shared multi-GPU systems, wasting valuable GPU memory capacity. Considering that each workload has highly varying memory demands, achieving high global memory utilization in multi-GPU system is a challenging problem.

2.2 Exploiting Neighbor GPU Memory

Modern GPU servers provide useful primitives to facilitate memory harvesting, i.e., leveraging fast intra-server GPU interconnects without modifying applications or frameworks.

Fast interconnect. Modern GPU servers are commonly equipped with 8~16 GPUs connected via high-speed interconnect such as NVLink. For instance, NVIDIA DGX-2 has 16 GPUs with direct peer GPU access through fully connected NVLink topology, yielding a large pool of 512GB GPU memory at 600GB/s GPU-to-GPU bidirectional bandwidth [23]. With this HW specification, modern GPU servers can provide an attractive option for GPU-to-GPU communication to

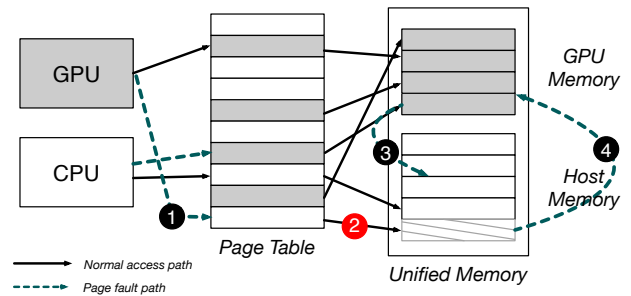


Figure 2: Unified address space in modern GPUs

transmit data without using the expensive PCIe interconnect.

Transparent control. Memory harvesting must address the cases that transfer data between GPUs running jobs across different applications. For example, in Figure 1, the graph analytics jobs and DL training jobs use their own frameworks. The unified virtual memory (UVM), introduced by NVIDIA and AMD, enables large-memory applications to seamlessly oversubscribe the limited GPU memory [29]. The technology is implemented as a GPU driver, so implementing memory harvesting in UVM requires no modification to the GPU applications or ML frameworks.

This section first explains how the current UVM driver works to leverage host memory for extending the limited GPU memory capacity. The UVM driver takes advantage of the host memory as a swap space for the GPU. Figure 2 presents how the UVM driver provides the unified address space across the GPU and host memory. The driver identifies whether the page being accessed by GPU is located on the GPU or the host memory through the page fault mechanism. With a single unified page table, UVM translates GPU virtual address into either GPU physical address or host physical address. If UVM identifies that the page accessed by the GPU kernel is mapped to the host by referring to the page table ①, a page fault exception is raised ② and the UVM driver brings the page into the GPU memory (typically via PCIe interconnect) ④. Meanwhile, when no free space is available in GPU, the driver needs to evict an old page from GPU memory before transferring the faulted page to GPU ③. Moving these pages requires page table entries to be properly updated to map new locations.

Opportunity. As observed in Figure 1, if one GPU needs more memory than its memory capacity, it can spill the fraction of oversubscription to one or more neighbor GPUs that still consume less than the total memory capacity with the fast interconnect. Moreover, by leveraging the unified address space supported by the UVM driver, we can serve diverse jobs ranging from graph analytics to DL training jobs without modifying applications or frameworks. Such *harvesting* of idle memory in neighbor GPUs opens up a unique opportunity to lower performance overhead under memory oversubscription by increasing the effective memory space. To work well

	PCIe	NVLink (speedup)
Throughput (GB/s)	12.3	40.1 (3.3×)
Latency (μ s)	16.7	5.1 (3.2×)

Table 1: Throughput and latency with PCIe and NVLink

with a very small fraction of idle memory, the mechanism for harvesting idle memory of neighbor GPUs should be timely and efficient. In Section 3.2, we address the design challenges in more detail.

3 Hierarchical Unified Virtual Memory

This section first measures the performance benefits of accessing the neighbor GPU’s memory connected through the high-speed interconnect (NVLink). Based on the measurements, we introduce a new unified memory, called *hierarchical unified virtual memory*, tailored for multi-layered memory systems comprised of local GPU, neighbor GPUs, and host memory. With the new memory organization, this section discusses how to incorporate the spare memory of neighbor GPUs into the memory hierarchy to bridge the performance gap between local GPU memory and host memory. Accessing neighbor GPUs is faster than accessing the host memory, but the spare memory space is dynamic and often limited, and may be shared by multiple harvesters. Therefore, utilizing a small amount of spare memory is crucial for effective harvesting. Finally, we discuss design challenges when leveraging the spare memory of neighbor GPUs with limited capacity.

3.1 Data Path with HUVM

To understand the performance benefit of harvesting the neighbor GPU memory, we conduct a performance analysis to measure the throughput migrating 2MB^2 data from a GPU memory to the host memory via PCIe and from a GPU memory to the other GPU memory via NVLink on an AWS `p3.8xlarge` instance. Two GPUs are connected through NVLink 2.0 with two lanes, providing up to uni-directional bandwidth of 50GB/s. Table 1 shows a comparison of bandwidth and latency between PCIe and NVLink. As expected, such NVLink provides more than $3\times$ better performance in terms of throughput and latency, indicating that data transfer time can be significantly reduced if we evict pages to neighbor GPUs. We anticipate that this performance gap will be higher in the next generation of high-speed interconnect.

Using the fast interconnect, i.e., NVLink, we build a new data path exploiting the spare memory of neighbor GPUs. Our approach brings two advantages in terms of performance. First, the new data path accelerates memory eviction. When evicting a memory chunk from the local GPU due to the

²Currently, the unit of memory eviction is a 2MB chunk in the UVM.

lack of memory space, if there is idle space in the neighbor GPU, our approach uses NVLink rather than PCIe to evict the memory chunk. Second, the new data path can reduce the latency of fetching. As the spare memory can act as a victim cache, we populate as many evicted chunks as possible on the spare memory. If these chunks are accessed again shortly, the chunks are fetched to the local GPU with the fast NVLink.

3.2 Design Challenges

Although our hierarchical unified virtual memory can reduce the performance penalty of GPU memory oversubscription, utilizing the spare memory poses several challenges.

- ① **Effective harvesting:** If the spare memory is not sufficient to serve all the evicted pages from the GPU applications, the effectiveness of memory harvesting would be limited to only buffering the evictions.
- ② **Minimal interference:** Harvesting may cause performance interference of the application running on the yielding GPU. Since we borrow the NVLink and PCIe bandwidth of the neighbor GPUs for the spare memory, our harvesting technique needs to minimize the performance interference.
- ③ **Low overhead:** Since UVM relies on the page fault mechanism, our approach inherits the same performance overhead, manipulating page table entries. Such overhead can prevent us from using full PCIe and NVLink bandwidth.
- ④ **Framework-agnostic:** All the GPU jobs do not rely on a specific framework. Our design and implementation need to be generalized to host a wide range of GPU workloads.

4 Memory Management for HUVM

This section presents a new memory manager, called *memHarvester*, for HUVM. *memHarvester* aims to hide the latency from the performance-critical path in accessing host memory with small but fast spare memory of neighbor GPUs. *memHarvester* opportunistically harvests the spare memory of neighbor GPUs while minimizing the performance penalty of memory-intensive applications that require more memory than available in one or more GPUs. To do so, *memHarvester* identifies the availability of spare memory in neighbor GPUs and plugs the spare memory into the memory hierarchy dynamically. To effectively harvest the spare memory, we explore a set of techniques: hierarchical and background eviction, fetching data in parallel, and prefetching in a neighbor GPU memory.

4.1 Overview

By harvesting spare memory in multi-GPU systems, *memHarvester* creates an illusion of GPU applications having a small cache between a GPU and host memory. Figure 3 presents the

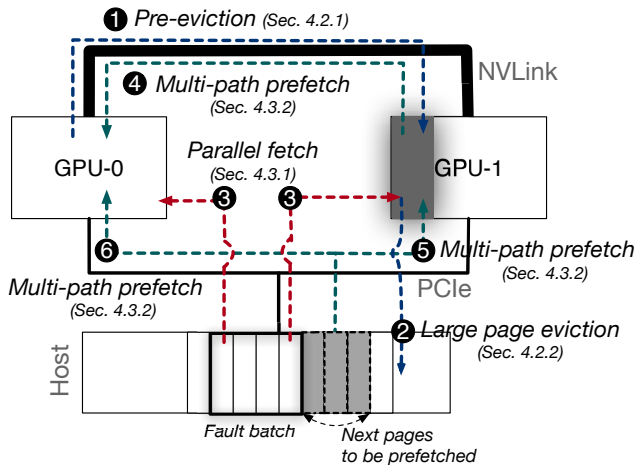


Figure 3: Exploiting spare memory with path diversity

overview of our proposed techniques. memHarvester allows GPU-0 to utilize the spare memory of GPU-1. ❶ Instead of evicting data to the host memory directly, memHarvester uses the spare memory connected through NVLink as a victim buffer to shorten the latency of memory evictions. To reduce the cost of memory evictions, memHarvester takes advantage of pre- eviction to the spare memory. With the spare memory, memHarvester reduces the number of evicting data to the host memory. However, it cannot eliminate accessing the host memory. ❷ To alleviate the penalty of populating pages on the host, we introduce a large page eviction scheme, amortizing the cost of evicting multiple base pages. ❸ If accessing data in the host memory is inevitable, we exploit the parallelism with path diversity in the multi-GPU systems when handling page fault batches. This approach utilizes the individual PCIe lanes attached to the harvested GPU and the local GPU at the same time.

To further hide the latency of migrating data to local GPU memory, we introduce a new *multi-path parallel prefetcher* exploiting the harvested memory and the path diversity in the multi-GPU systems. ❹ The data residing in the spare memory is prefetched into the local GPU through NVLink. Meanwhile, we prefetch the data belonging to the host memory to either the spare memory ❺ or local GPU memory ❻, depending on the PCIe congestion. Specifically, when there are multiple harvesters exercising the same spare memory, the harvested GPU can receive excessive prefetching requests, which saturate the PCIe bandwidth and adversely delay transfers of prefetch data. To address the problem, we have a facility dynamically enabling and disabling the use of spare memory in prefetching data from the host memory. Also, memHarvester prioritizes the memory eviction demands over prefetching on the harvested area because serving memory evictions is on the critical path of handling page faults.

Each GPU process has a separate page table. Even though evicted pages are located in the neighbor GPU yielding spare

memory, the process running in that neighbor GPU is not allowed to access the evicted pages. This is because the page table of the process in the neighbor GPU does not have the mapping information for the evicted pages, like the existing UVM driver [29]. Therefore, we do not violate the integrity and secrecy of evicted pages.

4.2 Hiding Eviction Latency to Host

Once the memory capacity is full, it is not allowed to bring data to the GPU before completing the memory eviction. The memory eviction is a part of the performance-critical path. Since migrating data to GPU memory is faster than to the host memory, our harvesting can reduce the latency of handling GPU page faults. After evicting the pages, memHarvester rapidly moves to the next step of fetching requested pages. While the harvesting GPU fetches the required pages, memHarvester invokes a background *writeback* thread to make a copy of the evicted page present in the harvested memory to the host memory. After copying pages, memHarvester marks the pages backed in host memory as *removable*. The purpose of the background copy is to immediately return the harvested space to the original GPU with negligible overhead. Once the application in the yielding GPU requires more memory than it currently has, it causes a GPU page fault. Then, memHarvester reclaims the harvested (*removable*) pages for the yielding GPU to use without the eviction. It picks the pages that come first into the yielding GPU. If there is no free or removable page, the yielding GPU may need to wait until the pages in the harvested region become removable. Although it can potentially incur the performance overhead, it rarely happens when evaluating our technique.

Therefore, our approach of using spare memory as a victim buffer allows that with a small fraction of memory, memHarvester turns the latency of host memory access into that of a neighbor GPU memory almost entirely, eliminating the host access latency from performance-critical eviction path.

4.2.1 Pre- eviction

When evicting pages to host memory, it is known that the pre- eviction scheme cannot hide the eviction latency entirely because the pre- eviction rate is limited to the PCIe bandwidth [25, 28]. In addition, pre- eviction requests contend with fetch requests occasionally, adding extra latency to critical fetch requests by stalling them. On the other hand, when evicting pages to harvested memory, it uses abundant NVLink bandwidth without contending the fetch requests from host memory. Once the memory consumption of harvesting GPU reaches a threshold of total physical memory (by default, if less than 50 free chunks are available), memHarvester invokes a pre- eviction thread. The pre- eviction has a good match with the background writeback technique because pre- eviction and writeback are pipelined. Ultimately, such pre- eviction allows

free memory available most of the time and effectively eliminates the eviction time from the GPU page fault latency. memHarvester uses the well-known LRU policy to select pages to be evicted. Pre-eviction with LRU policy works well with the memory access patterns of graphs and DNN workloads because most of them exhibit a cyclic memory access pattern with long reuse distances [2, 19, 35, 37, 38].

4.2.2 Large page eviction

The granularity for page faults is supposed to be the same size as the host architecture because the UVM driver relies on the demand paging scheme. On the other hand, the UVM driver uses a 2MB chunk as an eviction unit to simplify memory management. While evicting a 2MB chunk from the GPU to the host, the UVM driver splits the 2MB chunk into 512 4KB pages and performs the page population for the 512 pages because the driver is conservatively implemented to use the base page in the host architecture. To avoid such undesired inefficiency, memHarvester allocates 2MB of large pages in host memory by using the kernel's contiguous memory allocator [21]. Hence, memHarvester performs a single operation for populating a 2MB page between GPU and host memory instead of performing for individual 512 4KB pages. With or without our harvesting scheme, we can apply this large page eviction to all cases where a GPU has to interact with host memory.

4.2.3 Eviction policy

As multiple GPUs can leave a small amount of idle memory, as shown in Figure 1, memHarvester selects a target in a round-robin fashion to avoid hotspot contention and maximize the available GPU-to-GPU bandwidth in the system. Additionally, the round-robin policy enables each yielding GPU to make the removable pages in parallel through individual PCIe lanes. Eviction to the spare memory can incur performance interference to applications running on that yielding GPUs. We evaluate the negative performance impact of harvesting memory in Section 6.2.1.

When multiple harvesting requests are concentrated on a single spare memory, memHarvester handles the requests following their arrival orders. Note that the spare memory is used as a shared cache across harvesters.

4.3 Hiding Fetch Latency from Host

Evicting pages to the harvested memory reduces fetching latency if the local GPU accesses its evicted pages in a short period because memHarvester can fetch pages from the victim buffer in the neighbor GPU. In addition to that, memHarvester deploys two proactive schemes to hide fetch latency from the host memory with the limited space of harvested memory: fetching pages in parallel on page faults and pre-fetching pages with diverse paths.

4.3.1 Fetching pages in parallel

To reduce the cost of handling page faults, modern GPUs batch multiple page faults. The number of faults in a batch varies from time to time.³ The UVM driver handles requested pages corresponding to the page faults one by one. On the other hand, with the availability of harvested memory, memHarvester parallelizes handling multiple page faults in the same batch. memHarvester invokes page fault handling threads for each GPU (i.e., harvesting GPU and yielding GPUs), dividing tasks to each handling thread. As shown in Figure 3 (3), one thread for harvesting GPU takes faults from the head of the fault buffer, while the other thread for yielding GPU traverses the buffer from the tail and places the data corresponding to the fault on the harvested memory. memHarvester effectively reduces the latency of handling faults by fetching pages to local and the spare memory in parallel, so it utilizes the individual PCIe lanes attached to each GPU. Later, the fetched faults on the spare memory will be consumed through NVLink, reducing the fault latency further.

4.3.2 Multi-path parallel prefetcher

To hide the latency of accessing host and spare memory, we design a *multi-path parallel prefetcher* exploiting the path diversity in multi-GPU systems. When prefetching multiple memory chunks across the host and spare memory, there is no dependency between chunks. Our multi-path prefetcher can exploit the parallelism fetching the chunks with PCIe and NVLink. memHarvester first places the pages in the spare memory to the local GPU memory via NVLink (4 in Figure 3). For the pages in the host memory, memHarvester prefetches them on either the spare memory (5) or the local GPU memory through PCIe (6). The policy paragraph below explains how to select the target memory when prefetching from the host dynamically.

For selecting candidates to be prefetched, memHarvester uses a simple yet effective next line and stride prefetchers, which are good enough for graph analytics [2, 19, 38] and DNN training workloads [7, 32, 33, 35, 37]. By extracting the memory access pattern from the page fault history, memHarvester prefetches the next couple of chunks from either the host memory or the harvested memory, depending on where the chunks are located. We empirically select the amount of prefetch as 32MB and will show the sensitivity study in Section 6.2.1.

4.3.3 Prefetch policy

Unless the PCIe lane attached to the spare memory is crowded, we observe that prefetching to the spare memory can further reduce the fetch latency compared to the prefetch directly from the host to the local GPU. This is because leveraging

³The stock UVM driver handles up to 128 faults in a batch.

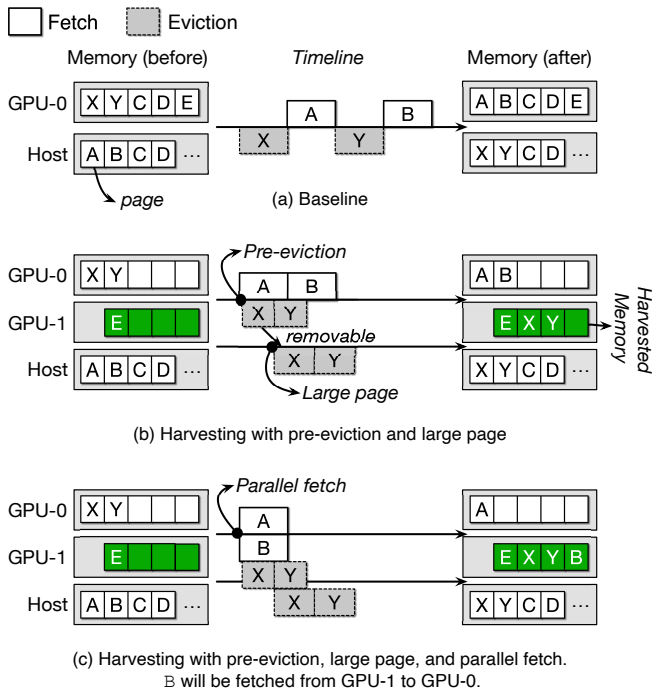


Figure 4: Timeline comparison for pre-emption, large page, and parallel fetch (Suppose that GPU-0 is harvesting the idle memory of GPU-1)

spare memory can reduce both 1) the number of page faults by proactively fetching pages and 2) the page fault latency by placing the pages highly likely to be accessed on the spare memory. On the other hand, as the number of active harvesters increases, the PCIe lane attached to the spare memory can be congested. Then, it slows down supplying the pages to the spare memory due to the limited PCIe bandwidth, increasing the fault latency.

To deal with diverse harvesting scenarios in multi-GPU servers, we have a policy in prefetching to dynamically select where the data in the host memory to be prefetched to either the spare memory (5) or the local GPU memory (6) based on the number of active harvesters.

4.4 Putting It All Together

Suppose the application running on the GPU-0 is accessing page A and B on the host memory. The GPU-0 memory is fully occupied except for the small number of reserved pages that hide memory eviction time in handling page faults. Figure 4a and Figure 4b compare how memHarvester eliminates the memory eviction latency from the critical path. While handling the page faults, if the number of the reserved chunks falls below the threshold (set as 50 chunks), memHarvester triggers the pre-emption task to secure the free space for upcoming page faults without the memory eviction. In this example, the oldest page X and Y in the GPU-0 are evicted

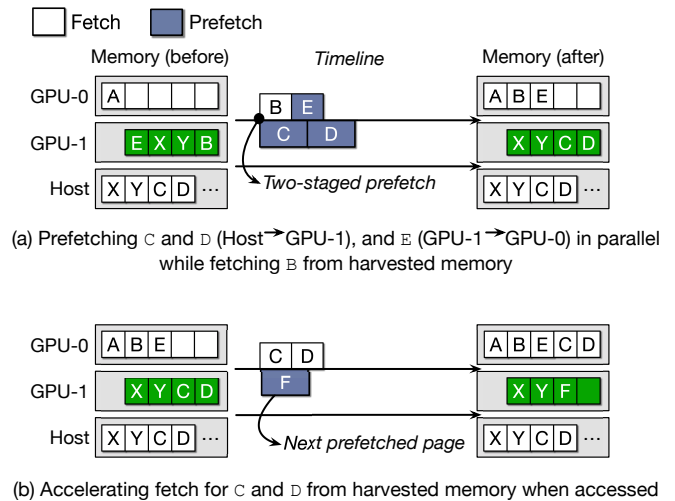


Figure 5: Timeline for our multi-path parallel prefetcher

to the harvested memory of the GPU-1 as background. Once it is completed, memHarvester makes a copy of that pages to the host and marks them as removable.

While handling the fault for page A, memHarvester looks up the faulted pages belonging to the same batch group. Figure 4c presents that page A and B are in the same batch. Thus, the fault batch can be processed to the local and harvested memory in parallel. While fetching page A to the local GPU memory, memHarvester places page B on the harvested memory. Since each GPU has its PCIe lane, we can fetch page A and B in parallel. Although it cannot reduce the number of page faults, we can hide the latency for fetching page B. Eventually, it does reduce the fault latency by fetching page B from the harvested memory connected through NVLink, rather than the host via PCIe.

Since the page A and B are faulted in order, memHarvester prefetches the next page C, D, and E. Assume that the number of pages to be prefetched is three in this example. Figure 5a depicts how the multi-path parallel prefetcher works. As page C and D are in the host memory, those can be prefetched into either the harvested memory or the local GPU memory through the PCIe lane. In this example, we assume that the PCIe lane is not contended so that memHarvester selects the harvested memory as the target. On the other hand, page E, which is in the harvested memory, is prefetched to the local GPU memory in parallel via NVLink. It can eliminate the fault if page E is accessed from the application. Figure 5b presents that we can potentially reduce the fault latency for page C and D by fetching such pages from the harvested memory when they are accessed.

5 Implementation

We implement our prototype, memHarvester, in the NVIDIA UVM driver version 460.67. The modification of the UVM

driver is 1,838 lines of C code measured by SLOCCount. We do not require any modifications to runtime and frameworks.

5.1 Managing Spare Memory

As UVM, memHarvester manages GPU physical memory as 2MB chunks. Each chunk has metadata, which describes the states and physical address of the 2MB chunk. For each GPU, memHarvester maintains a linked list of the metadata for free 2MB chunks. By referencing the free list of GPUs, memHarvester can easily extract available spare memory in the system. Once a GPU harvests a neighbor GPU's memory, memHarvester marks the metadata to indicate that a 2MB chunk is harvested from the neighbor GPU.

5.2 Managing GPU Memory Eviction

To evict a chunk, memHarvester selects the oldest chunk from per-GPU LRU⁴ lists as the stock UVM. The pre-eviction path diverges depending on the availability of harvested memory. If memHarvester has harvested memory, the background thread evicts chunks to the harvested memory. Otherwise, it evicts chunks to host memory. When memHarvester evicts a chunk to the harvested memory, memHarvester moves the chunk metadata from the LRU list to the evicted list. Since the evicted chunk resides only on the harvested memory, memHarvester does not allow the chunk to be reclaimed by the harvester. Once the eviction to the harvested memory is completed, memHarvester invokes a writeback thread to duplicate the chunk in the harvested memory to host memory. After that, memHarvester marks the chunk as removable and moves the chunk to the removable list. When the harvested memory has to be returned to the original GPU, memHarvester reclaims the removable chunks first. Before completing the writeback task, we are not able to reclaim the harvested space for serving the other request. Thus, the throughput of the writeback thread is critical. memHarvester increases the eviction size with large page to accelerate writeback thread and eviction.

5.3 Managing Fetch Requests

When a page fault exception occurs, the fault exception handler supplies faults from the head of the fault batch as usual. At the same time, memHarvester wakes up another kernel thread to serve fault entries from the tail of the fault batch to the harvested memory in parallel. As a result, we spend less time completing the fault batch with harvested memory. To coordinate consuming the fault batch shared between two threads, we use the mutex lock to handle a fault entry from the buffer synchronously.

After handling the demand fault batch, memHarvester triggers our multi-path prefetcher utilizing both the PCIe and NVLink bandwidth. memHarvester employs a simple but

⁴The LRU term presents the least recently swapped-in page.

effective next-line prefetcher for capturing the memory access pattern observed in graph analytics and DNN training. It keeps track of the addresses for the faulted pages to identify the direction of previous page fault addresses. Once the direction is determined, memHarvester prefetches the next 32MB as default. For the selected chunks, we examine the metadata to filter out the chunks already in the local GPU. We separate the selected chunks into two different queues according to their origin, either the host or the harvested memory, and then conduct the multi-path prefetch. While prefetching the chunks in the host to the harvested memory through PCIe, we use a kernel thread to prefetch the other chunks in the harvested memory to the local GPU via NVLink. We mark prefetched chunks as removable so that it can be immediately reclaimed when needed. If we encounter on-demand faults while prefetching, we abandon the ongoing prefetches to serve the demand faults first.

6 Evaluation

6.1 Experimental Setup

To evaluate the effectiveness of memHarvester, we conduct performance comparisons with the stock version of the unified virtual memory (Base) and the prior approach employing the pre-eviction and prefetch techniques for the host memory [11, 14, 17, 25, 28] (Pre-ef-host). As the implementation of prior studies is not publicly available, we imitate the behavior of pre-eviction and prefetch on top of the stock UVM driver. The evaluation is performed on an AWS p3.8xlarge which has four NVIDIA V100 GPUs, each with 16GB of memory. These four GPU cards are connected to each other through NVSwitch and NVLink 2.0 [16] and 240GB of host memory is attached through PCIe 3.0.

6.2 Experimental Results

6.2.1 Inter-job Harvesting

First, we evaluate our scheme in a shared multi GPU server hosting multiple DNN training and graph analytics workloads. We show how the idle memory of GPUs can be harvested by memory-intensive workloads running on other GPUs with inter-job harvesting. In this evaluation, all training workloads run with PyTorch (version 1.10.1), and all graph analytics workloads run with cuGraph (version 21.12).

Performance improvement. We evaluate the execution time of multiple workloads in four scenarios by varying the type of jobs and the number of harvesters to mimic a shared multi-GPU server environment. Table 2 presents the scenarios and the memory usage ratio for each job according to the given graph dataset or batch size. Figure 6 shows the speedup of the execution time to Base and also measures the performance

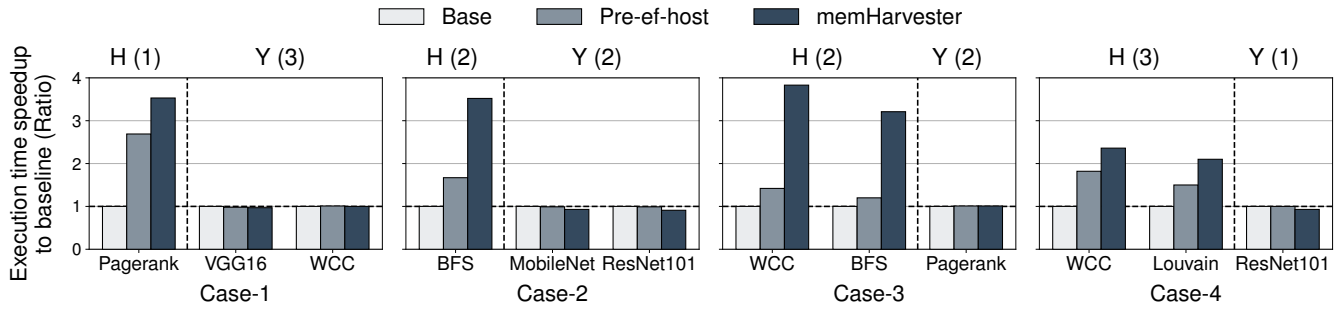


Figure 6: Execution time speedup of memory intensive workloads with memHarvester on four different harvesting scenarios (H: Harvesting GPU, Y: Yielding GPU, and the numbers in parentheses indicate the number of participating GPUs)

	GPU-0	GPU-1	GPU-2	GPU-3
Case-1 (ratio)	Pagerank (1.87x) soc-twitter-2010	VGG16 256 (0.96x)	WCC (0.79x) soc-sinaweibo	
Case-2 (ratio)	BFS (1.51x) web-uk-2005	MobileNet 256 (0.91x)	ResNet101 64 (0.67x)	
Case-3 (ratio)	WCC (1.77x) soc-twitter-2010	BFS (1.70x) soc-twitter-2010	Pagerank (0.74x) web-Clue09-50m	
Case-4 (ratio)	WCC (1.77x) soc-twitter-2010	Louvain (1.47x) web-uk-2005	ResNet101 64 (0.74x)	

Table 2: Multi-job scenarios with memory usage ratio in parentheses, input graph, and batch size (Gray cell: harvester)

of Pre-ef-host (i.e., the prior approach [17, 28]) for performance comparison. For all the cases, the execution time of harvesters, which do not fit on one or more V100 GPU memory (16GB), can be significantly improved by harvesting the idle memory of neighbor GPUs connected through NVLink.

In Case-1, Pagerank running on GPU-0 benefits from the spare memory of GPU-1 and 2 where VGG16 is running in data-parallelism and spare memory of GPU-3 where WCC is running, leading to 3.53 \times and 1.31 \times improvement over Base and Pre-ef-host, respectively. The amount of total spare memory of GPU-1, 2, and 3 is around 4.64GB, which is much smaller than the overcommitted memory of 13.92GB by Pagerank. Although the spare memory is unable to serve all the evictions from Pagerank, our memHarvester effectively harvests the spare memory to hide the latency of accessing the host. In addition, the negative performance impact to the applications running on the yielding GPUs is negligible.

We also evaluate the performance changes by varying the number of harvesting and yielding GPUs. In Case-2, BFS running on GPU-0 and 1 harvests the spare memory of GPU-2 and 3 where MobileNet and ResNet101 are running in each of the GPUs. memHarvester considers that the separated spare memory is logically unified like a shared cache across the harvesters. Our memHarvester can increase the performance of BFS by 3.52 \times and 2.1 \times compared to Base and Pre-ef-host, respectively. While harvesting, the perfor-

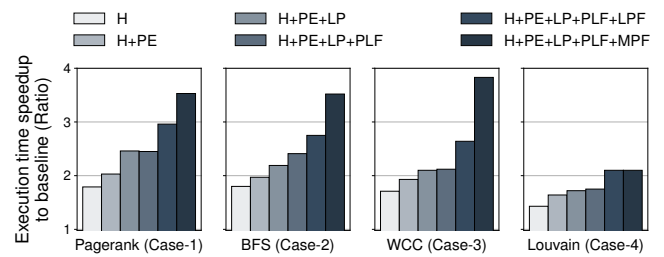


Figure 7: Effectiveness of individual techniques

mance impact of yielding GPUs is around 7~9%.

We show a different scenario in Case-3 where two applications each of which runs on a single GPU contend for the idle memory of neighbor GPUs. WCC running on GPU-0 and BFS running on GPU-1 harvests the spare memory of GPU-2 and 3 where Pagerank is running with its graph partitioned across two GPUs. Our memHarvester can increase the performance of WCC by 3.83 \times and 2.71 \times , and that of BFS by 3.21 \times and 2.67 \times compared to Base and Pre-ef-host, respectively. There is no performance degradation in Pagerank running on the yielding GPU.

In Case-4, we evaluate the effectiveness of our approach when two workloads share a limited spare memory contributed by a single GPU. ResNet101 yields 4.16GB idle memory that is shared by WCC running on GPU-0, and Louvain running on GPU-1 and 2. Although the throughput improvement is not much compared to other cases, it still outperforms pre-ef-host by 30~40%. Compared to Base, the harvesters show around 2.1~2.36 \times improvement while the performance impact of yielding GPU is around 7%.

Analysis of performance improvement. We decompose the contribution of the performance improvement to individual schemes constituting memHarvester. To this end, Figure 7 shows performance changes for each workload (from the left to the right) while we enable spare memory harvesting (H), pre-eviction (PE), large page support (LP), parallel fetch (PLF), local prefetcher (LPF) and multi-path parallel prefetcher (MPF) in order. Note that local prefetcher (LPF) is not a scheme of

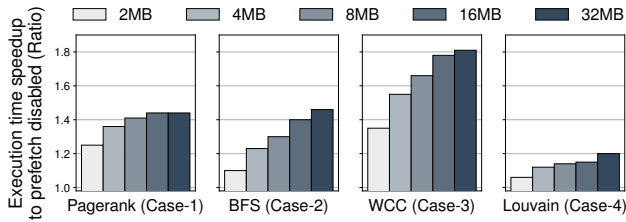


Figure 8: Sensitivity to the amount of prefetch

memHarvester and it is evaluated as a baseline to compare the performance gain of our multi-path parallel prefetcher (MPF).

In general, we observe higher performance gain while we enable each scheme one by one. This is because each scheme has its complementary benefit to memHarvester: i) spare memory harvesting (H) utilizes the spare memory as an eviction buffer and a victim cache to reduce the latency of migrating chunks by using NVLink rather than PCIe; ii) pre-eviction (PE) eliminates the eviction latency from the critical path by reducing on-demand page faults; iii) large page support (LP) reduces the time of making removable pages by writing back the chunks to host in batch; and iv) parallel fetch (PLF) reduces the latency of handling on-demand page faults by fetching the pages in the fault batch in parallel with both PCIe and NVLink. While the first three schemes (H, PE, LP) focus on optimizing the eviction penalty, parallel fetch (PLF) focuses on reducing the latency when handling page faults.

Finally, we investigate our multi-path parallel prefetcher (MPF) which drastically improves performance compared with local prefetcher (LPF). For Case-1, 2, and 3, some of the evicted data reside in the host memory due to the lack of aggregated idle GPU memory in the system. Our multi-path parallel prefetcher utilizes the PCIe of the yielding GPU to prefetch data in host to the harvested memory. For Case-4, however, our multi-path parallel prefetcher (MPF) selects to prefetch data in the host memory to the local GPU memory rather than the harvested memory to avoid contention in the PCIe lane attached to the yielding GPU. Because there is more than one harvester per one spare memory, multi-path parallel prefetcher (MPF) changes the policy to directly fetch data from the host to local GPU memory. Thus, multi-path parallel prefetcher (MPF) has no performance gain compared to local prefetcher (LPF) in Case-4.

Sensitivity study. In this subparagraph, we present the sensitivity study of memHarvester to examine three aspects.

① **Prefetch size and stride:** We evaluate the sensitivity study for our next line and stride prefetches used in multi-path parallel prefetcher. Figure 8 depicts the execution time improvement by varying the amount of prefetches from 2MB to 32MB with a 2MB stride. Since both graph analytics and DNN training workloads exhibit sequential access patterns during the execution [2, 19, 35, 37, 38], our prefetcher extracts the direction of accessing the memory address and issues the

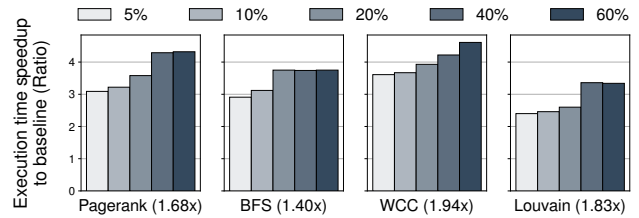


Figure 9: Sensitivity to the size of spare memory (The numbers in parentheses indicate the overcommitment ratio)

prefetch requests for the next chunks to either the host memory or the spare memory depending on where the selected chunk is located (host to spare and spare to local). While evaluating our prefetcher, we observe that the 2MB stride shows better throughput than the next line (0MB stride). For all four cases, increasing the amount of prefetch steadily improves the performance. We select 32MB as our default prefetch size with a stride of 2MB. We will further investigate how the amount of prefetch can be dynamically adjusted for maximizing the prefetch effects in our future study.

② **Available spare memory:** In a shared multi-GPU server, we expect that the available memory for harvesting fluctuates. We evaluate the execution time of four graph analytics workloads by manually varying the amount of spare memory from 5% to 60%. We imitate a scenario with 2 GPUs with one GPU running a memory-intensive graph analytics workload and the other GPU yielding spare memory with an appropriate size of `cudaMalloc`. Figure 9 exhibits that the performance can be improved by harvesting more idle memory of neighbor GPUs. Interestingly, even with 5% (800MB) of spare memory, we can achieve more than $2\times$ improvement for all four workloads. By effectively managing the small amount of idle memory of a neighbor GPU, the performance improvement is significant. However, when a certain amount of spare memory is harvested to accommodate all the active working sets to fit in the local GPU with the harvested memory, maximum performance is achieved and there is no additional improvement even if we increase the amount of spare memory. We observed that the amount of overcommitment size is larger than the active working set. For Louvain with an overcommitment ratio of $1.83\times$, the maximum performance is achieved when the spare memory size is 40% and even if the spare memory size is increased, there is no additional performance gain. We found a discrepancy between the size of the active memory working set and the size of memory `malloc'`d by the user-level and will further investigate whether we can also harvest the unused memory space that is not included in the active working set in our future study.

③ **Performance interference:** While harvesting spare memory, our approach can cause performance interference to the applications running on GPUs that yield the spare memory. This is because the harvesters piggyback the

	ResNet101			
	Base	H	H+PE+LP	memHarvester
VectorAdd (1)	1	0.99	0.99	0.97
VectorAdd (3)	1	0.91	0.88	0.87

Table 3: Normalized execution time of ResNet101 by increasing the number of VectorAdd harvesters

memory and PCIe bandwidth of yielding GPUs. Table 3 presents the impact of performance interference through a VectorAdd microbenchmark designed to generate a significant memory harvesting traffic. By increasing the number of VectorAdd harvesters, we measure the performance of ResNet101, which yields idle memory on one GPU. To investigate the interference incurred by individual schemes, we decompose our schemes into harvesting only (H), with pre-emption and large page (H+PE+LP), and with all the prefetchers (memHarvester). When running with a single harvester, the impact of performance interference for ResNet101 is negligible, up to 3% compared to the baseline. The maximum pressure to the GPU yielding idle memory is bounded by the network bandwidth of two GPUs through NVLink.

Meanwhile, three harvesters reduce performance by 13%. Even with harvesting only (H), it shows 9% performance degradation. As the number of harvesters increases, the amount of memory traffic to the idle memory can also increase, leading to the memory bandwidth contention. When enabling pre-emption and large page (H+PE+LP) schemes, it can utilize the idle memory more effectively, but it poses an additional 3% overhead. When all the proposed schemes are applied (memHarvester), the performance interference is not much different. Since our multi-path parallel prefetcher (MPF) selects to prefetch data in the host memory to the local GPU memory rather than the harvested memory to avoid contention in the PCIe lane attached to the yielding GPU. We anticipate that throttling the harvesting traffic can reduce the performance interference though it decreases the benefits to the harvesters. We leave this optimization to future work.

6.2.2 Intra-job Harvesting.

Our memory harvesting technique can be applied to multi-GPU applications where the memory consumption of individual GPUs is not even. The representative example is multi-GPU DNN training exploiting pipeline parallelism [20]. Such memory usage imbalance is primarily due to non-identical model partitions placed across the GPUs to balance the computation of each partition, leading to different memory demands (e.g., some of the GPUs need to have multiple weight and activation versions in pipeline parallelism). We evaluate the effectiveness of HUVm with memHarvester for single DNN training jobs. For this evaluation, we select GNMT16, GNMT8, ResNet50, and VGG16 with PyTorch. For the baseline,

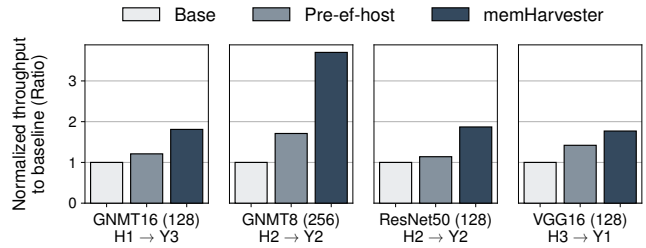


Figure 10: Throughput improvement for single training workloads (H: # harvesting GPU and Y: # yielding GPU)

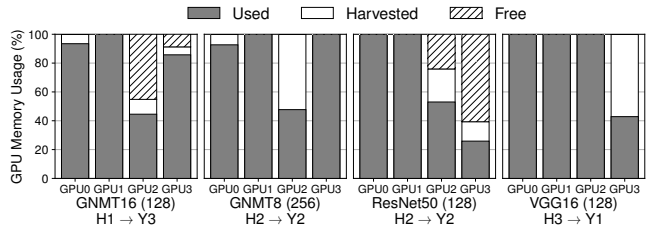


Figure 11: Memory usage for single training workloads (H: # harvesting GPU and Y: # yielding GPU)

the PyTorch framework is modified to support the memory oversubscription with NVIDIA UVM.

Performance improvement. Figure 10 shows the results of throughput comparison among memHarvester, Pre-ef-host, and Base. To initiate memory oversubscription, the batch size in each model is chosen such that at least one of the four GPUs goes beyond the local memory capacity. The figure shows that for all the models, memHarvester outperforms Pre-ef-host. In particular, for GNMT16 and ResNet50, memHarvester can effectively eliminate the host memory accesses in the increased batch size by harvesting only idle memory of neighbor GPUs. This leads to the throughput in memHarvester 1.5~1.6 \times higher than that in Pre-ef-host for the two models.

Figure 11 shows the memory profiles of the four models across the individual GPUs. It is worth noting that model training under memHarvester achieves throughput gains via diverse memory harvesting paths. For GNMT16, GPU-1 is the only GPU harvesting the spare memory of GPU-0, 2, and 3. On the contrary, for ResNet50, GPU-0 and 1 are the two harvesting GPUs that utilize the spare memory of the other two yielding GPUs, i.e., GPU-2 and 3.

More interestingly, for GNMT8 and VGG16, the aggregated idle memory across the yielding GPUs is not sufficient to serve the data evicted from the harvesting GPUs. In VGG16 with batch size 128, GPU-0, 1, and 2 use up all the idle memory of GPU-3 and then require using the host memory additionally. In spite of exercising the host memory, memHarvester shows meaningful throughput gains for both workloads. memHarvester improves throughput over Pre-ef-host by 2.16 \times and 1.24 \times for GNMT8 and VGG16, respectively.

7 Related Work

To the best of our knowledge, memHarvester is the first to propose a framework that allows GPU applications to utilize neighbor GPU's memory connected through the high-speed interconnect (NVLink). There have been significant efforts in both the architecture and systems community to support GPU memory oversubscription while minimizing the performance degradation of applications. Demand paging on GPUs [3, 22, 24] allows the GPUs to move pages from the GPU's memory to/from the CPU's memory automatically. We survey recent techniques that provide mechanisms to allow applications with a large working set to run on GPUs.

Framework-guided approach. For deep learning (DL) training workloads, prior studies proposed to have the framework insert the pre- eviction and pre- fetch operations by analyzing the dataflow graph [11, 17, 28]. Peng et al. introduced a sophisticated technique employing the pre- fetch and recomputation opportunistically without relying on the dataflow graph [25], these also require the intensive framework modification in terms of tensor allocation. Besides the pre- fetch technique, Animesh et al. proposed to compress the data, which shows the long reuse distance to save memory while the data is not being actively used [12]. Although this design approach can be effective, it requires the framework modification and understanding of the target applications, incurring the engineering overhead. Unlike such previous studies, we design and implement our solution in the GPU driver which can coordinate all the GPU memory in a centralized way.

Architectural approach. The architecture community also explored hardware techniques to minimize the overhead of GPU memory virtualization. Recently, Choukse et al. explored the advantage of leveraging the neighbor GPU memory which is connected through the high-bandwidth interconnect (NVLink). Unlike our approach, they studied a HW-based compression scheme to squeeze the limited neighbor GPU memory [5]. Ganguly et al. studied the prefetch technique used in the NVIDIA UVM driver in the overcommitted environment and proposed a HW-based pre- eviction and pre- fetch techniques [8]. Kim et al. exploited that modern GPUs handle the page faults in a batch and co- designed the GPU runtime and hardware to overlap the page eviction and migration effectively [14]. Li et al. proposed a framework for memory over- commitment [17] to efficiently virtualize GPU memory, but the disadvantage of those studies requires significant changes to the GPU runtime and hardware. Although such hardware approaches can further improve the performance as well as the efficiency, it requires new hardware structures.

Memory compression. Many previous studies [5, 15, 17, 27, 30, 34] proposed techniques to perform memory compression. While these techniques allow more data on the GPU memory, they are orthogonal to HUVm and can be used in conjunction to further improve the effectiveness of our proposal.

8 Conclusion

In this study, we propose a new approach of virtualizing multi-GPU memory, hierarchical unified virtual memory, by dynamically incorporating the spare memory of neighbor GPUs in multi-GPU systems. This can alleviate the memory fragmentation problem by creating the illusion of GPU applications having an increased effective memory space. To effectively utilize the small fraction of neighbor GPUs' memory, we introduced a memory manager for multi-GPU systems that has a set of techniques, including large page support, parallel fetch, and multi-path parallel prefetcher. Since our techniques can effectively reduce the latency of accessing host memory, for memory-intensive workloads, throughput performance is significantly improved compared to baseline and prior studies based on pre- eviction and prefetch techniques.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their valuable comments and feedback. This research is supported by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(2021-0-02051) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation) and Electronics and Telecommunications Research Institute(ETRI) grant (22ZS1300). This work is also supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. 2020R1C1C1014940).

References

- [1] B. Acun, M. Murphy, X. Wang, J. Nie, C. Wu, and K. Hazelwood. Understanding training efficiency of deep learning recommendation models at scale. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.
- [2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyong Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.
- [3] AMD. Radeon's Next-generation Vega Architecture. <https://en.wikichip.org/w/images/a/a1/vega-whitepaper.pdf>, 2017.
- [4] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Amer Jaleel, Carole-Jean Wu, and David Nellans. Mcm-gpu: Multi-chip-module gpus for continued performance scalability. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

- [5] E. Choukse, M. B. Sullivan, M. O'Connor, M. Erez, J. Pool, D. Nellans, and S. W. Keckler. Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.
- [6] cuGraph. GPU Graph Analytics. <https://github.com/rapidsai/cugraph>.
- [7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805*, 2018.
- [8] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019.
- [9] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A gpu cluster manager for distributed deep learning. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.
- [10] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Bolland. Graphie: Large-scale asynchronous graph traversals on just a gpu. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.
- [11] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
- [12] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.
- [13] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.
- [14] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Haddi, and Hyesoon Kim. Batch-aware unified memory management in gpus for irregular workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
- [15] J. Kim, M. Sullivan, E. Choukse, and M. Erez. Bit-plane compression: Transforming data for better compression in many-core architectures. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016.
- [16] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Trans. Parallel Distributed Systems (TPDS)*, 31(1), January 2020.
- [17] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.
- [18] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.
- [19] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus. *Proc. VLDB Endow.*, 2020.
- [20] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.
- [21] Michal Nazarewicz. A deep dive into cma, Mar. 2012. <https://lwn.net/Articles/486301/>.
- [22] NVIDIA. NVIDIA Tesla P100 P100 GPU Architecture. <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>, 2016.
- [23] NVIDIA. Nvidia dgx-2: The world's most powerful ai system for the most complex ai challenges., 2019. <https://www.nvidia.com/en-us/data-center/dgx-2/>.

- [24] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture. <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf>, 2020.
- [25] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
- [26] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.
- [27] M. Rhu, M. O’Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. Keckler. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.
- [28] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vdn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Processing of the 49th International Symposium on Microarchitecture (MICRO)*, 2016.
- [29] Nikolay Sakharnykh. Maximizing unified memory performance in cuda, Nov. 2017. <https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/>.
- [30] V. Sathish, M. Schulte, and N. Kim. Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.
- [31] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. Graphreduce: processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.
- [32] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations (ICLR)*, 2015.
- [33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NIPS)*, 2017.
- [34] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. Mowry, and O. Mutlu. A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015.
- [35] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
- [36] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
- [37] Peifeng Yu and Mosharaf Chowdhury. Fine-grained GPU sharing primitives for deep learning applications. In *Proceedings of Machine Learning and Systems (MLSys)*, 2020.
- [38] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *Proceedings of the IEEE International Conference on Big Data (Big Data)*, 2017.



Zero Overhead Monitoring for Cloud-native Infrastructure using RDMA

Zhe Wang¹, Teng Ma², Linghe Kong¹, Zhenzao Wen², Jingxuan Li², Zhuo Song², Yang Lu²
Yong Yang², Tao Ma², Guihai Chen¹, Wei Cao²
¹Shanghai Jiao Tong University
²Alibaba Group

Abstract

Cloud services have recently undergone a major shift from monolithic designs to microservices running on the cloud-native infrastructure, where monitoring systems are widely deployed to ensure the service level agreement (SLA). Nevertheless, the traditional monitoring system no longer fulfills the demands of cloud-native monitoring, which is observed from the practical experience in Alibaba cloud. Specifically, the monitor occupies resources (*e.g.*, CPU) of the monitored infrastructure, disturbing services running on it. For example, enabling monitor causes jitters/declines of online services in Alibaba’s “double eleven” shopping festival with high loads. On the other hand, the quality of service (QoS) of monitoring itself, which is vital to track and ensure SLA, is not guaranteed with the high loaded system.

In this paper, we design and implement a novel monitoring system, named ZERO, for cloud-native monitoring. First, ZERO achieves zero overhead to collect raw metrics from the monitored hosts using *one-sided* remote direct memory access (RDMA) operations, thus avoiding any interferences to cloud services. Second, ZERO adopts receiver-driven model to collect monitoring metrics with high QoS, where credit-based flow control and hybrid I/O model are proposed to mitigate network congestion/interference and CPU bottlenecks. ZERO has been deployed and evaluated in Alibaba cloud. Deployment results show that ZERO achieves no CPU occupation at the monitored host and supports 1 ~ 10k hosts with 0.1 ~ 1s sampling interval using single thread for network I/O.

1 Introduction

Recent shifts in the production cloud environment from monolithic designs to microservice-based architecture [33, 34] have made cloud-native infrastructure the cornerstone of cloud computing services. The cloud-native applications consist of thousands of single-concern, loosely-coupled microservices running on containerized platforms [76]. The underlying systems are treated as disposable and immutable, finally enabling highly available, flexible and scalable cloud services.

In order to ensure the service level agreement (SLA) [52], the whole infrastructure is monitored with not only the upper-layer application metrics, but also the fundamental system metrics [84]. The novel cloud-native infrastructure, however, brings new challenges/demands to cloud-native monitoring, along with two major issues in commercial deployments.

First, traditional monitoring systems [10, 11, 65] occupies host (physical/virtual machine, PM/VM) resources to collect, process and upload metrics (Figure 1), which inevitably causes resource contentions with cloud services — enabling monitors causes jitters/declines of online services in Alibaba “double eleven” shopping festival (Figure 3). To ensure service SLA with resource constraints, the deployed monitor at the host should have no resource occupation.

Second, the quality of service (QoS) of monitoring itself is not guaranteed, which fails to support massive metrics with rapid variations in cloud-native monitoring. The latency/throughput of monitoring jitters severely due to the high system loads or small CPU quota set by the cloud provider (Figure 4). However, monitoring system with high QoS is vital to track and ensure SLA of monitored services [80, 84].

To resolve the limitations of traditional monitoring system and fulfill the demands of cloud-native monitoring, we design and implement a novel ZERO monitoring system in this paper. ZERO proposes a receiver-driven model, which collects raw metrics from the monitored host via *one-sided* RDMA operations, *i.e.*, RDMA read. Based on the ZERO framework, the monitoring system is expected to achieve no CPU occupation at the monitored host, low latency and high throughput, finally avoiding any interferences to services and fulfilling the QoS requirements of large-scale distributed monitoring.

However, there still exist several challenges to achieve the above goals. As shown in Figure 1, traditional monitor collects and processes raw metrics from the monitored processes, then upload metrics to the remote host, which inevitably causes CPU occupations. How to manage memory regions of system/application metrics and expose them to the remote host, finally achieving zero-overhead monitoring via RDMA read, is challenging. On the other hand, the remote

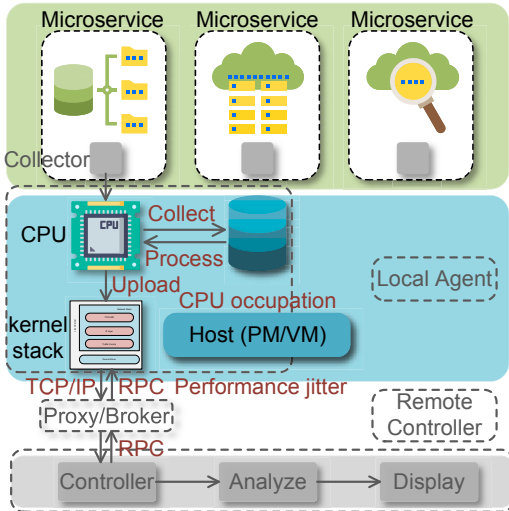


Figure 1: Traditional monitoring System.

monitoring host becomes bottlenecks in the receiver-driven model, as a tradeoff of offloading local monitoring overhead to the remote host. The remote host performs RDMA read on many monitored hosts, resulting in incast problem [85]¹. The remote host not only collects metrics, but also processes raw metric for further operations, all of which are CPU intensive. How to enable large-scale monitoring with network/CPU bottlenecks is challenging as well.

To access raw metrics with no CPU occupation, ZERO proposes the novel control plane and data plane. For the ease of clarity, we separate ZERO into local agent and remote controller (Figure 2). To achieve high scalability in reliable connection (RC) mode [26, 82], system/application metrics are managed by one agent and share one queue pair (QP) connection. In the control plane, ZERO agent provides universal interfaces for systems/applications to register the memory regions of their metrics at the RDMA NIC (RNIC). The metadata of these metrics are recorded at the control region. ZERO controller can thus acquire metadata of metrics from the control region as the prerequisite to access raw metrics. All metrics only need to register once if the metadata is not updated, after which ZERO agent enters blocking mode. In the data plane, the memory regions of metrics (data region) are exposed to the agent process via shared memory, finally to the remote controller. The ZERO controller can thus perform RDMA read on the data region directly without involving memory copies and CPU usages at the monitored host. As a result, ZERO achieves disposable overhead in the control plane and zero overhead in the data plane.

To deal with the network/CPU bottlenecks at the controller, ZERO proposes credit-based flow control (Credit-FC) and hybrid I/O model. We observe that the receiver-driven model is

¹Incast problem happens when multiple senders transfer data to one receiver simultaneously.

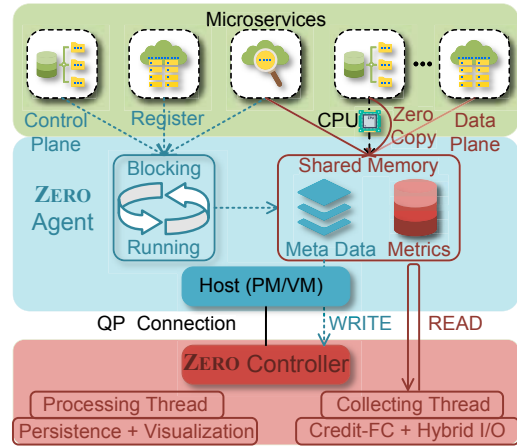


Figure 2: ZERO monitoring system.

superior to the traditional sender-driven model (*i.e.*, agent actively uploads metrics as per heartbeat): i) collecting metrics on demand to ensure the QoS of monitoring on latency; ii) limiting the total in-flight data of concurrent flows to avoid network congestion/interference. Accordingly, ZERO proposes Credit-FC to mitigate the incast problem while fulfilling the latency/throughput requirements of monitoring. On the other hand, ZERO introduces hybrid I/O model (a combination of event driven and busy polling mechanisms) and adopts thread dispatching to remedy the CPU bottlenecks of collecting and processing metrics, respectively.

As case studies, we integrate application (Redis [12]) metrics, system (kernel/containers [27, 76]) metrics and eBPF [3] metrics into the ZERO framework to demonstrate its generality and flexibility. We also share our experience about building large-scale monitoring system using RDMA.

The major contributions of this paper are summarized as follows:

- We propose the first zero-overhead monitoring system, ZERO, to resolve limitations of traditional monitoring system in cloud-native monitoring.
- We tackle several challenges of zero-overhead monitoring, including data plane with no CPU involvement, network congestion/interference caused by monitoring traffics, and CPU bottlenecks at the controller.
- We have deployed and evaluated ZERO in Alibaba cloud. ZERO achieves no CPU occupation at the monitored host and supports 1 ~ 10k hosts with 0.1 ~ 1s sampling intervals. We also share our experience with ZERO.

The paper is organized as following. Section 2 introduces the background and motivation. Section 3 proposes zero-overhead monitoring. Section 4 designs and implements ZERO framework. Section 5 presents case studies. Section 6 evaluates the proposed design. Section 7 introduces the experience and future work. Section 8 discusses related works and Section 9 concludes this paper.

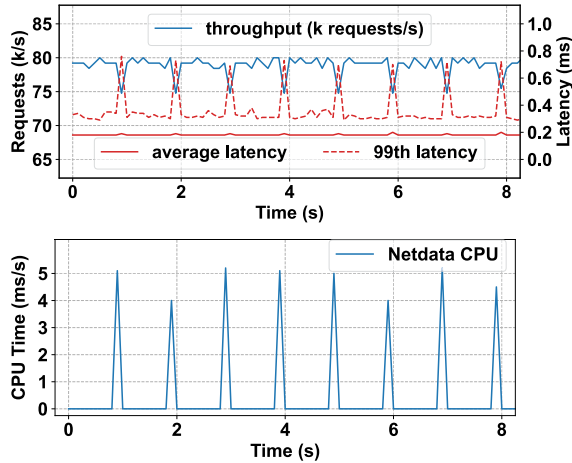


Figure 3: Monitor interfering with services.

2 Background and Motivation

In this section, we further elaborate cloud native monitoring and the inherent limitations of traditional monitoring system.

2.1 Cloud-Native Monitoring

Monitoring system deployed at the cloud-native infrastructure, namely, cloud-native monitoring, is indispensable to ensure the SLA of cloud services. Monitor collects the bottom-layer system metrics, such as the utilization of physical resources (CPU, memory, *etc.*). Based on system metrics, monitoring system performs health checks on the underlying system, makes early alert and provides suggestions to administrators [18]. Furthermore, by analyzing the historical resource consumption and performance variations, cloud providers improve system utilization and lower operational expenses (OpEx) [27, 28, 41]. On the other hand, the upper-layer application metrics, *e.g.*, requests per second of key-value service [12, 45], directly reflect user activities and functional state of applications. The monolithic applications are decoupled to thousands of microservices [33], all of which are monitored to track and ensure the SLA.

The novel cloud-native applications together with the essential infrastructure bring new challenges/demands to cloud-native monitoring, along with two major issues in commercial deployment.

How to avoid monitor interfering with services? Microservices have much stricter requirements of QoS compared with typical applications [33]. However, the cloud-native environment is highly resource constrained. For example, Alibaba cloud adopts mixed deployment of CPU-intensive online service [42] and I/O-intensive batch jobs [93] at the same host, to maximize resource utilization [43, 78] and reduce long-term capital expenses (CapEx). Microsoft Azure also reports that 80% of VMs only have 1 ~ 2 vCPU cores [27]. The monitor

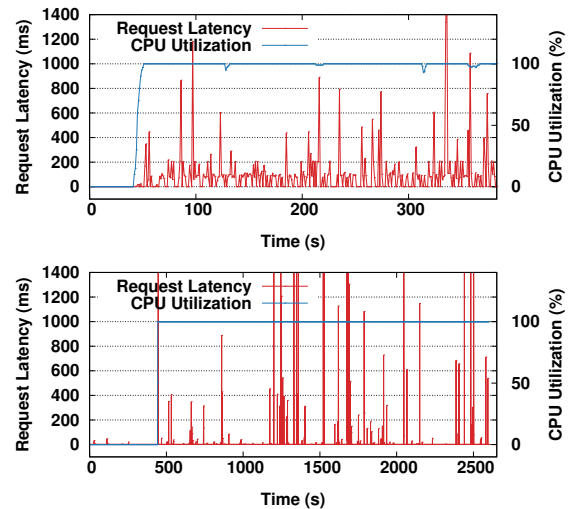


Figure 4: Monitoring jitters with high system loads in Netdata (upper) and Prometheus (bottom).

should have no CPU occupation to avoid contentions with cloud services.

How to ensure QoS of monitoring? Cloud-native monitoring needs to support massive metrics with rapid variations. Cloud providers, such as Alibaba, Netflix and Uber, need to monitor millions of metrics with hundreds to thousands of microservices [80]. Besides, these application metrics (financial transactions, social network and e-commerce [67, 94]) and system metrics (CPU, memory, network [6, 54]) have rapid variations with a time scale of seconds/milliseconds. To track and ensure SLA of services, monitoring requires high QoS from the perspective of latency and throughput.

2.2 Traditional Monitoring System

We next introduce traditional distributed monitoring systems [10, 51, 65] and elaborate their limitations as the motivation of ZERO. As shown in Figure 1, multiple collectors acquire application/system metrics via specific interfaces, meanwhile raw metrics are processed as the final outcomes (Section 5). After that, the collected metrics are uploaded to the remote controller for further analysis and visualization. Each step occupies host CPUs for memory copy, calculation, network transmission, *etc.*, which inevitably interfering with services running on the host. On the other hand, traditional monitor massively relies on the kernel's TCP/IP network stack to transmit metrics. However, kernel data processing overhead has become the main bottleneck of end-to-end latency/throughput [90]. State-of-the-art works thus offload network functionality from kernel to user-space network stack [39, 53, 64] or hardware [35, 44, 70].

We next elaborate the limitations of traditional monitors in our real deployment. We consider two representative open-

sourced monitors, i.e., Netdata [10] and Prometheus [11]. We observed that enabling monitors causes jitters of online service in Alibaba “double eleven” shopping festival with high loads. We use Netdata to monitor a high-loaded host running Redis services with 1s sampling interval. As shown in Figure 3, monitor process causes jitters of Redis service, i.e., the throughput declines by 6.25% while tail latency increases by $2\times$ periodically, due to the CPU occupation in each monitoring cycle. We analyze that reasons for such “interference spikes” are two folds. First, the deployment of service and monitor processes may adopt default CPU scheduling or specific CPU bonding [48], where contentions happen when service/monitor processes are scheduled/bonded to the same CPU core. Second, the CPU utilization keeps on high water level and exhibits burst natures, especially during sales promotion with high loads. Thus, the duty-cycled monitoring process with slight CPU occupation ($1 \sim 5\%$) already causes severe interference (Figure 3). The CPU breakup of monitor shows that the uploading phase occupies $5 \sim 10\%$ of the total CPU utilization while the collecting phase occupies the majority. On the other hand, the QoS of monitoring is highly affected by the system load or CPU quota set by the cloud operator. The latency of monitoring increases by more than $10\times$, when host CPU is saturated or CPU quota is reached (Figure 4). Assigning dedicated cores for monitoring may avoid these problems, however, causes large wastes of resources and high CapEx. Besides, the monitoring process may be blocked to request metrics via service interface (Section 5).

3 Zero-overhead Monitoring

To avoid resource contentions with services and ensure QoS of monitoring, we propose the zero-overhead monitoring system, namely ZERO, for cloud-native monitoring. ZERO exploits features of monitoring metrics and RDMA capability in modern data center. The basic idea is that most of raw metrics are counters updated at fix memory regions — the remote controller can thus obtain these metrics by performing RDMA read on these memory regions — without any CPU involvement at the monitored side. We next elaborate possibilities and challenges of realizing ZERO.

Metric features. ZERO is based on two features of monitoring metrics. First, most of the monitoring metrics are counters. Systems/applications update these metrics at fixed memory region after initialization. For instance, the number of stored/evicted key-value pairs in Redis and the number of sent/received packets recorded by kernel stack are all counters. Second, processing of raw metrics is simple algebraic calculation and can be offloaded to the controller. For example, Redis exports statistic data on the raw metrics and per-CPU counters are summed to get the final kernel metrics. These features are general to system/application metrics and ZERO can thus support various metrics as an universal framework (Section 5).

RDMA support. ZERO then leverages *one-sided* RDMA operation to read raw metrics/counters, to achieve no CPU/kernel involvements at the monitored host. RDMA has been widely used in data centers and provides new characteristics of low latency (as low as $1 \mu s$), high bandwidth (more than 100Gbps) and kernel/CPU bypass. RDMA supports both *one-sided* and *two-sided* operations. The one-sided operations directly operate on the remote memory via *read* and *write* without involving the remote server’s CPU. To perform one-sided RDMA operations, one needs to register the *memory region* (MR) at the RNIC of remote host and acquire the generated *remote protection key* (rkey). The two-sided operations, i.e., *send* and *recv*, communicate via an interface similar to socket. In the following paper, we refer to RDMA read, write, send and recv as READ, WRITE, SEND and RECV, respectively. RDMA hosts create queue pairs (QP) consisting of a send queue and a receive queue, then post RDMA operations on send/receive queue to communicate with the remote host. RDMA transport supports reliable or unreliable connection (RC/UC) and unreliable datagram (UD). One-to-one connections between QPs are required in RC/UC mode, whereas one-to-many communication is supported in UD mode. Different transport types support different subsets of RDMA operations, and READ operation is only supported in RC mode.

Challenges. While ZERO is expected to achieve zero-overhead monitoring, there still exist two challenges to make the idea practical. First, we observe that most CPU time of traditional monitor are spent on collecting metrics from system/application processes (Section 2.2). ZERO also needs to eliminate such overheads in its data plane, besides the transmission overheads offloaded to RNIC. Second, controller is bottlenecked on both network and CPU with large number of monitored hosts, as all monitoring overheads are offloaded to the remote controller. With all these challenges, the key innovation of Zero lies in effectively exploiting one-sided RDMA and designing the separate control/data plane to realize zero-overhead monitoring. Zero further incorporates several designs to resolve practical issues (network congestion/interference, scalability) in distributed monitoring.

4 Design and Implementation

In this section, we present the overview of the ZERO framework. We then introduce the design and implementation of ZERO in details.

4.1 Overview

As shown in Figure 2, ZERO proposes the novel control plane and data plane to collect raw metrics without CPU involvements at the monitored host. ZERO adopts receiver-driven model to collect metrics from large number of hosts, and deals with the network and CPU bottlenecks at the controller.

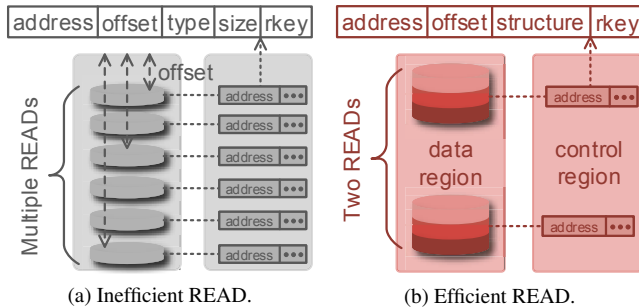


Figure 5: READ w/o (left) or w/ (right) memory management.

The scalability of RDMA-based system is constrained by the on-chip memory (SRAM) of RNIC [26, 82]. To achieve high scalability at the controller, all metrics at the host are managed by one agent and share one QP connection. In the control plane, ZERO agent provides interface for systems and applications to register the memory region of their metrics (data region) at the RNIC. The metadata of metrics (*e.g.*, address, rkey) are written into the control region. ZERO controller can thus obtain metadata by reading the control region and access to raw metrics via reading the data region. The agent process is blocked unless systems/applications need to register/update metrics. In the data plane, the memory region of metrics are exposed to the agent process via shared memory, finally to the remote controller. The controller performing READ on the data region to acquires metrics, which achieves zero copy and no CPU involvement at the monitored host. In the real deployment, the control-plane overhead is usually disposable — the metrics only need to register once — then keeps in use or updates after a long time. The data plane has no CPU occupation as expected.

ZERO supports large-scale monitoring via single controller. ZERO adopts thread dispatching, where only $1 \sim 2$ threads are used to collect metrics and the rest cores are used to process metrics in parallel for further operations, *e.g.*, visualization and persistence. ZERO controller uses receiver-driven model, *i.e.*, issuing READ requests on the monitored hosts to collect raw metrics, which is superior to the traditional sender-driven model. ZERO achieves high monitoring QoS by posting READ requests on demand and avoids network congestion/interference by controlling the total in-flight data. Accordingly, credit-based flow control (Credit-FC) and hybrid I/O model are proposed to remedy the incast problem and CPU bottleneck, respectively.

4.2 ZERO Control Plane and Data plane

We introduce the ZERO control plane and data plane, together with ZERO framework usage and interface in this subsection.

Control plane. ZERO agent deals with registration requests from systems/applications, which uses UNIX domain socket for inter-process communications (IPC). To handle the request, the MRs of monitoring metrics are registered at the

RNIC and the metadata of MRs (*e.g.*, address, type, size, rkey) are recorded at the control region (Figure 5a). Note that the MR is pinned after metrics are registered and will be released only after metrics expire. ZERO agent registers the control region at the RNIC and builds a QP connection with the remote controller in advance. The controller can thus acquire the metadata of metrics by reading and parsing the control region, then access raw metrics.

The control plane has disposable overhead. ZERO agent inevitably occupies host resources to deal with registration requests. However, we observe that the control-plane overhead is disposable for most of metrics. We divided metrics into three types according to the practice in Alibaba cloud. First, metrics of the underlying systems and persistent services, *e.g.*, database and storage services [56, 91], are usually immobile once registered. Second, some services may be dynamically created/destroyed. For example, e-commerce services are periodically expanded/shrunk according to the number of users shopping online [60]. Third, some user requests are served by ephemeral serverless *functions* to mitigate the cost of long-lived services with intermittent activities [25, 33].

Generally, ZERO agent enters into blocking mode with no CPU occupation (Figure 2). When services change, ZERO agent resumes to (de)register metrics and update the corresponding control region. For the first two cases, ZERO agent handles registration requests and updates control regions infrequently with negligible overhead. ZERO controller also only reads control regions once in a long period. In the serverless case, ZERO agent may frequently (de)register metrics of serverless *functions*. ZERO agent further uses WRITE to write the updated metadata into the remote control region (Figure 2). ZERO controller can thus obtain raw metrics by only reading the data region with one RTT. Note that the overhead of ZERO using WRITE or SEND/RECV is still much lower than that of the traditional monitor (Section 6.2).

Data plane. To eliminate the overhead of collecting metrics from multiple processes and avoid frequent memory copies, ZERO exclusively adopts shared memory in its data plane. Specifically, ZERO agent uses `mmap` operations, which are invoked when registering metrics, to expose MRs of metrics (data region) to the agent process, finally to the remote controller. The `mmap` operation takes 4KB page as the basic unit. However, metrics are not necessarily locating at the page header (Figure 5). ZERO agent calculates the page header of metrics to `mmap` their pages. As shown in Figure 5, ZERO agent records the page address and page offset in the control region. To protect data region from being modified by local agent or malicious remote host, ZERO sets read-only access to the data region via `mmap` flags and uses the rkey mechanism inherently supported by RDMA. ZERO controller obtains raw metrics by performing READ on the data region.

The data plane has zero overhead. The data plane achieves zero CPU occupation, zero copy, and no extra memory footprint at the monitored host, via the shared memory design.

```

// type one, specifying attributes of variables
struct disk my_disk{
    .disk      = "sda",
    .hash      = 0x000f3456, ...
} __attribute__((section(".zero_init")));
//type two, using allocator
struct disk *my_disk = zero_malloc(sizeof(struct disk));

```

Figure 6: Management interface for two types of metrics.

An alternative solution of copying metrics when updating causes frequent CPU occupations for memory copies, and extra memory footprints. Besides, ZERO data plane ensures the read-write consistency between remote and local memory. Most of application/system metrics are defined as atomic variables, which are updated atomically in the shared memory. The atomic update only needs $1\sim 3ns$ in Intel Haswell architecture [74] — three orders of magnitude lower than that of RDMA operations ($1\mu s$) — the memory consistency is guaranteed between update and READ. For non-atomic variables, ZERO uses bit flags to indicate the states of updating. FaRM [31] and Pilaf [68] use a similar method to ensure data consistency with READ. ZERO controller will check whether metrics are read correctly via bit flags and retry in the next cycle. We eliminate all synchronous locks for zero overhead, which may cause inconsistency under rare race conditions while ensuring accuracy for most cases.

Memory management is indispensable. While local host achieves zero overhead in ZERO data plane, we observe that simply performing READ on massive metrics cannot achieve desirable performance. Because metrics are distributed across the process/kernel space with discrete memory addresses, ZERO controller needs to read large number of entries in control region as well as metrics in data region (Figure 5a). However, the bandwidth of READ falls rapidly and the latency is nearly doubled when the number of MRs increases from 100 to 10k, due to evictions in the RNIC SRAM [26, 82]. ZERO introduces memory management to reduce the number of MRs and READ requests required to collect massive metrics. Specifically, ZERO proposes two memory-management mechanisms for two types of metrics (Figure 6). First, many metrics are global variables or data structures. One can mark these metrics by specifying attributes of variables [13]. The compiler will distribute these metrics to the same data segment. Second, metrics are defined as pointers to variables. ZERO provides a memory allocator for these metrics. Specifically, the MRs of metrics are allocated with continuous space via the allocation API. The core idea of both methods is concatenating metrics to the same MR to support massive metrics. Besides, data region is aligned as `struct` and recorded at the control region for the ease of memory parsing at the controller. As shown in Figure 5b, ZERO controller only needs to post one READ request on the same MR to get a list of metrics.

Framework usage and interface. ZERO can be easily deployed at hosts (PM/VM) with RDMA support. ZERO agent and controller need to be initiated at the local and remote host respectively. Systems/applications then invoke agent API to manage and (de)register their metrics. The agent accomplishes all control-plane operations, e.g., `mmap`, updates of control region, when handling (de)register requests.

4.3 Scaling-out Monitoring

We next present how to support large-scale monitoring with single ZERO controller. ZERO proposes credit-based flow control (Credit-FC) and hybrid I/O model, to avoid network congestion/interference and remedy CPU bottlenecks respectively. ZERO controller needs to collect and process metrics from large number of hosts, while fulfilling the monitoring QoS in latency and throughput. To achieve this goal, the controller adopts thread dispatching to collect and process metrics in parallel with individual threads.

Collecting metrics. The controller only assigns $1\sim 2$ threads to collect metrics. ZERO achieves high efficient network I/O with single thread by posting READ requests then polling completions on multiple QPs in batch. This is feasible because both `post_send` and `poll_cq` are fast non-block operations. According to our experiment, the batch operations only add negligible latency (tens of μs).

Receiver-driven model is superior to send-driven model. Issuing READ on data region turns out a receiver-driven model to collect metrics from multiple hosts. The receiver-driven model has two benefits compared with the traditional sender-driven model. The controller posts READ requests on demand to meet the target latency or updating frequency in monitoring. Besides, it also facilitates to avoid network congestion/interference by limiting the total in-flight data of concurrent flows. We next intuitively formulate the scale-out ability of such receiver-driven model. The monitored hosts have different requirements in terms of updating interval U and data size S , i.e., controller needs to collect S bytes in every U seconds for a specific host. Assuming the bandwidth B of the receiver is fully utilized, the maximum number of supported hosts $n = B \times U / S$. Our deployment shows that ZERO supports at least 1k hosts with 128KB metrics and 100ms sampling interval.

Credit-based flow control. Concurrent READ requests generate burst network traffics, which are transmitted from multiple hosts to the controller simultaneously, resulting in severe incast problems. ZERO introduces Credit-FC to remedy the incast problem, which works as follows.

Large-sized READ requests are segmented into fix-sized fragments with 4KB page size. ZERO chooses such moderate size due to three considerations: i) page is the basic unit of shared memory, which can accommodate 1k 32-bit metrics and fulfill the demands of most services; ii) the number of READ requests is bounded as the total size of metrics in

single host is general hundreds of KBs; iii) the small size facilitates congestion control in severe incast.

Subsequently, credits are used to limit the total in-flight data of concurrent flows (identified by a QP). Posting READ requests or polling completion events will consume or regain credits for the target flow. The state-of-the-art works [44, 70] adopt bandwidth-delay product based flow control (BDP-FC), which bounds the in-flight data per flow by the BDP of the network. However, BDP is large enough to cause congestion and trigger explicit congestion notification (ECN) packets or priority-based flow control (PFC) pause frames [95], with large number of concurrent flows (Figure 11c). ZERO proposes Credit-FC to limit the total in-flight data. Specifically, the credit of each flow C_f is set to T/n , where T is the total credit and n is the number of concurrent flows. Finally, Credit-FC effectively avoids triggering ECN/PFC (Section 6.3) and network interference with service traffics (Section 7).

Hybrid I/O model. To avoid the thread performing network I/O being saturated, ZERO proposes the hybrid I/O model incorporating event driven and polling mechanisms, which is similar to NAPI in Linux [73]. The event-driven I/O can effectively avoid CPU occupation for busy polling. Each QP is associated with an event channel to notify a new (first) completion event. The I/O multiplexing interface, *e.g.*, `epoll`, is used to listen the `fds` of multiple event channels. The collecting thread blocks until completion events are notified from one or multiple QPs, then polling the in-flight requests of the corresponding QPs. We observe that the event-driven model effectively reduces CPU utilization with large number of in-flight requests (Figure 12b). However, when the monitored data size is too small with only several requests after segmentation, the `epoll` syscall and thread blocking incur high variations in READ latency (Figure 12a). The controller thus uses busy polling for hosts with small number of requests. In the hybrid I/O model, ZERO assigns two threads to perform event-driven polling and busy polling, respectively. Note that both threads share Credit-FC.

Processing metrics. The controller dispatches multiple threads to processing raw metrics in parallel. Specifically, the MRs of the collected metrics are placed into the appropriate queue where each MR can be handled by one of multiple processing threads. Each MR concatenated by a list of metrics is parsed as `struct` directly according to the metadata recorded at the control region. The parsed metrics are then processed by reproducing the same calculations which are originally performed by the monitored host. Finally, the controller imports processed metrics into InfluxDB [7] for persistence and uses Grafana [5] for visualization.

5 Case Study

In this section, we present how to integrate application/system metrics into the ZERO framework using three typical cases of

Redis [12], Linux kernel [81] and eBPF [3].

Redis Case. Redis [14] has been widely deployed in Alibaba cloud as database, cache, and message broker, providing low-latency in-memory data structure storage services. Traditional monitor acquires Redis metrics by requesting `INFO` interface of Redis server. Traditional monitor thus occupies the resource of Redis server and the host to obtain metrics. As a comparison, ZERO only needs to register metrics and requires no resource occupation for collecting metrics. There exist more than two hundred metrics in each Redis service instance. The naive implementation is performing READ on these metrics one-by-one (see Figure 5a), resulting in high latency and CPU utilization. To resolve this problem, we use allocation API (type two) to allocate and structuralize Redis metrics with continuous memory. As shown in Figure 5b, ZERO only needs to register and perform READ once for each Redis instance in our implementation. ZERO controller then parses metrics as `struct` according to the memory structure.

Linux Kernel Case. Linux kernel exports system metrics to user space via `proc` interface, which creates files under `/proc` directory and bonds corresponding kernel functions. Traditional monitor usually needs to read hundreds of `proc` files to get all system metrics, which incurs extra overhead for user/kernel-space processing. With ZERO framework, kernel metrics are registered at the ZERO agent then exported to the ZERO controller directly. We use ZERO to monitor metrics managed by container namespace for the container-based services [1]. Kernel metrics are usually implemented as lock-free per-CPU counters to avoid locking overhead. ZERO controller needs to READ all replications of metrics in each CPU core, locating at separate pages. Production cloud environment adopts fine-grained resource assignment and isolation, in which more than 90% hosts only have 1 ~ 4 CPU cores [27]. ZERO only needs 1 ~ 4 requests to obtain kernel metrics.

eBPF Case. The extend Berkeley Packet Filter (eBPF) [3, 66] is an evolving technology, which can dynamically attach program to running kernel for tracing, instructing, and even controlling the kernel code path. eBPF has been widely used in cloud computing for monitoring [6, 54], networking [38, 83], virtualization [21, 22] and security [29, 30]. We use eBPF to monitor traffics and retransmissions of large number of TCP connections in MaxCompute [32, 93] service. eBPF provides in-kernel data structure, called `map`, to enable control and data messages delivery within kernel or between kernel and user space. eBPF attaches probes to kernel/application functions at runtime and exports metrics, events and histograms to eBPF `map`. User process reads the entry of eBPF `map` via syscall. However, reading large number of entries incurs large overhead due to frequent syscalls. To integrate eBPF into ZERO framework, eBPF `array map` is adopted, which supports `mmap` operations (from Linux kernel 5.5+) and can thus export its memory to ZERO agent directly.

Name	Nodes	Hosts	OS kernel	Intel Xeon CPU code	Mellanox NIC	Protocol	ECN	PFC
Cluster1	65 × PMs	1024 × VMs	Linux 5.5	E5-2682 (64 cores)	2 × 25GbE ConnectX-4 Lx	RoCEv1/2	✗	✓
Cluster2	9 × PMs	1024 × Containers	Linux 3.10	Platinum 8369B (64 cores)	200GbE ConnectX-6 Dx	RoCEv1/2	✓	✓

Table 1: Deployment environment.

6 Evaluation

6.1 Evaluation Setup

In our evaluation, we adopt a multi-phase deployment with two typical clusters, as summarized in Table 1. Initially, we deploy Zero in a test environment (Cluster1) with a rational scale that mimics the production environment for demonstration. We next deployed ZERO in a public-cloud environment (Cluster2), which covers common cloud services in production. The deployment scale has continued to grow according to the feedback of canary testing and the actual demands of services. In Cluster1, services operates in guest VMs with 4 vCPU cores. The RNICs are virtualized and assigned to VMs via passthrough [87]. In Cluster2, services are deployed in containers running on bare-metal servers [92]. Each VM/container is monitored independently to evaluate the scalability of ZERO. Note that both configurations are typical in Alibaba cloud-native platform [15]. We use ZERO to monitor typical services, *e.g.*, Redis [14], container [1] and MaxCompute [32, 93].

We evaluate ZERO performance from three aspects:

- *CPU Utilization*: The CPU utilization is defined as occupied CPU time per second. We use `perf` tool to measure the CPU utilization of both ZERO agent and controller. We verify the CPU involvement of monitor in control plane and data plane. We also need to concern about the CPU utilization of ZERO controller for scalability.
- *Latency*: The latency of ZERO is the time used to READ all metrics from the monitored host. For traditional monitor, the latency consists of time used to collect/process all metrics and time used to upload all metrics. We verify whether monitor can meet up the updating frequency of metrics by latency.
- *Throughput*: The throughput is the collected bytes per second during each monitoring period. We verify whether monitor can support massive metrics by throughput.

We test the impact of the following parameters on monitoring performance:

- *Sampling Interval*: The required sampling interval is determined by the updating frequency of metrics. We evaluate ZERO with 10 ~ 1000ms sampling intervals and use 1000ms by default.
- *Number of Instances/Connections/Requests*: All of three parameters impact the CPU utilization and data size of monitoring. The number of service/container instances varies from 10 to 40, with a default value of 10. The number

Metric	Monitor	Redis	Kernel	eBPF
Total Latency (ms)	Baseline	0.7 ~ 19.3	0.5 ~ 1.6	0.8 ~ 12.5
	ZERO RPC	0.08 ~ 0.18	0.14 ~ 0.36	0.10 ~ 1.02
	ZERO	0.05 ~ 0.14	0.07 ~ 0.23	0.08 ~ 0.87
Agent CPU Utilization (%)	Baseline	0.5 ~ 45	0.01 ~ 4	0.08 ~ 6
	ZERO RPC	0.01 ~ 0.55	0.08 ~ 0.9	0.05 ~ 0.68
	Control plane	0.05 ~ 0.07	0.8 ~ 1.5	0.04 ~ 0.05
	Data plane	0	0	0

Table 2: Summary of ZERO overhead.

of TCP connections varies from 1k to 16k, with a default value of 1k. The number of work requests varies from 8 to 128 for a host, with a default value of 32.

- *Number of Hosts*: One ZERO controller is deployed to monitor multiple hosts running ZERO agent. We increase the number of hosts (from 64 to 1024) to evaluate the scalability of ZERO framework.

We use the state-of-the-art monitoring system named Netdata [10], existing *NX* monitor in Alibaba cloud, and ZERO RPC as comparison benchmarks:

- *Netdata*: Netdata is widely used by cloud providers, *e.g.*, Amazon Web Services (AWS) and Microsoft Azure [8]. Netdata integrates application/system metrics in one agent by requesting application interface or reading `proc` files respectively. The agent uploads metrics to the controller as per heartbeat or the controller acquires metrics via sending RPC requests to the agent.
- *NX*: *NX* is a network monitoring tool deployed in Alibaba cloud. *NX* exports network metrics as logs, *e.g.*, info of TCP connections, then uploads the collected metrics via log service. We have integrated ZERO framework into the *NX* monitor to improve its performance.
- *ZERO RPC*: ZERO RPC adopts the same data plane to access raw metrics. However, ZERO RPC is implemented via SEND/RECV instead of READ. The controller issues RPC requests to the agent, after which agent returns metrics as response. We explore design alternations of two-sided RDMA via ZERO RPC.

6.2 ZERO Overhead

We evaluate the CPU utilization and latency of ZERO in monitoring Redis, container and MaxCompute services. We present overall performance and summarize two key observations, followed by detailed micro-benchmarks for each case.

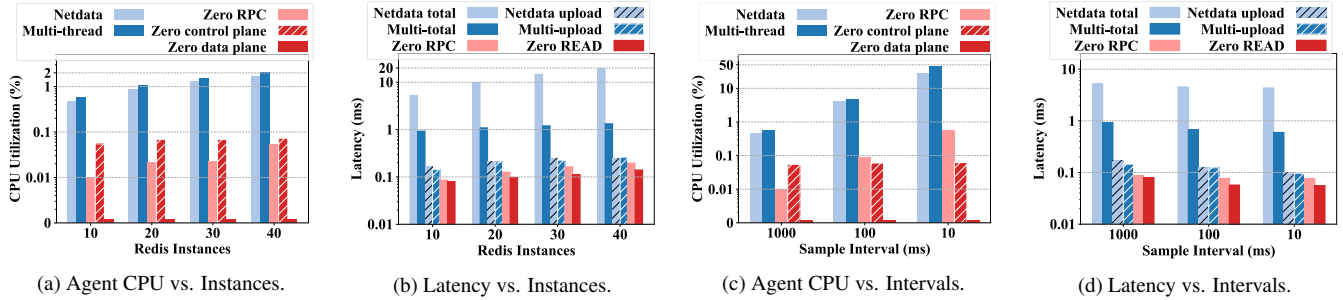


Figure 7: Monitor performance with 10 – 40 Redis instances and 10 – 1000ms sampling interval.

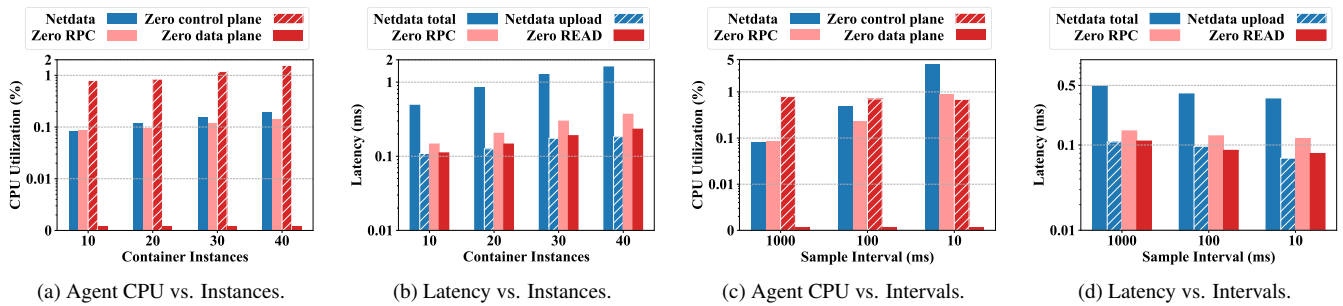


Figure 8: Monitor performance with 10 – 40 container instances and 10 – 1000ms sampling interval.

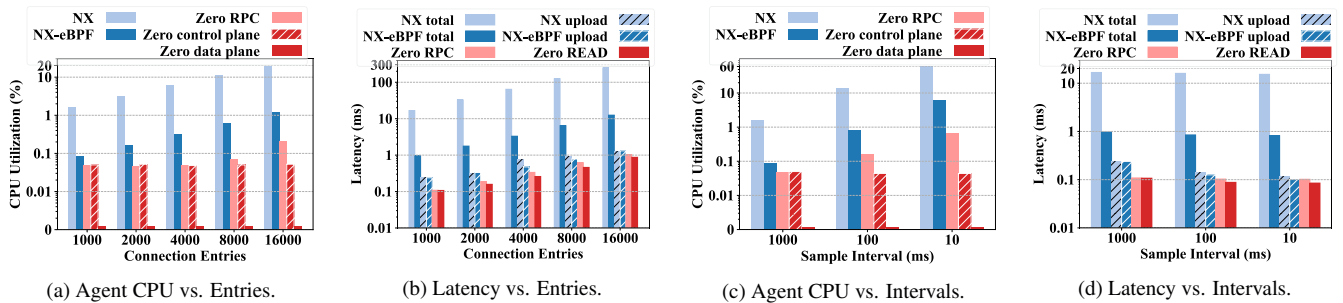


Figure 9: Monitor performance with 1 – 16k TCP connections and 10 – 1000ms sampling interval.

Overall. Table 2 summarizes the overhead of ZERO monitor. We focus on the monitoring latency and the CPU utilization of ZERO agent. Both Netdata or NX are referred as baselines. First, ZERO monitor reduces latency by one/two order of magnitudes compared with baselines. Our following breakdown of total latency reveals that baseline methods spend most of time to collect metrics from system/application processes. The TCP-based baselines actively upload metrics and achieve similar latency in uploading phase as ZERO READ. ZERO RPC has higher latency than ZERO READ because each RPC requires at least two RTTs [45, 68]. Second, ZERO agent achieves disposable overhead in control plane and zero overhead in data plane. The CPU utilization of ZERO control plane only increases slightly when registering more MRs, which is not affected by the sampling interval. The CPU utilization of ZERO data plane is always zero as expected. On the contrast,

the CPU utilization of baselines reaches very high values with lower sampling intervals. ZERO RPC eliminates the overhead of collecting metrics, however, the CPU utilization for posting SEND/RCV requests and polling completions still increases with lower sampling intervals. In summary, the benefit of ZERO to cloud services is enabling higher SLA of infrastructure, which effectively avoids performance jitters caused by CPU interference. Zero also improves monitoring performance, which reduces latency by 1~2 orders of magnitude and increases throughput by 3~6 \times (Section 6.3).

Redis Case. The monitoring performance of Redis case is shown in Figure 7. Netdata uses single or multiple threads to collect metrics from multiple Redis instances. With the increment of service instances, the CPU utilization of single- and multi-thread Netdata increase from 0.4 ~ 0.5% to 1.5 ~ 2%, which already severely interferes Redis services (Figure 3).

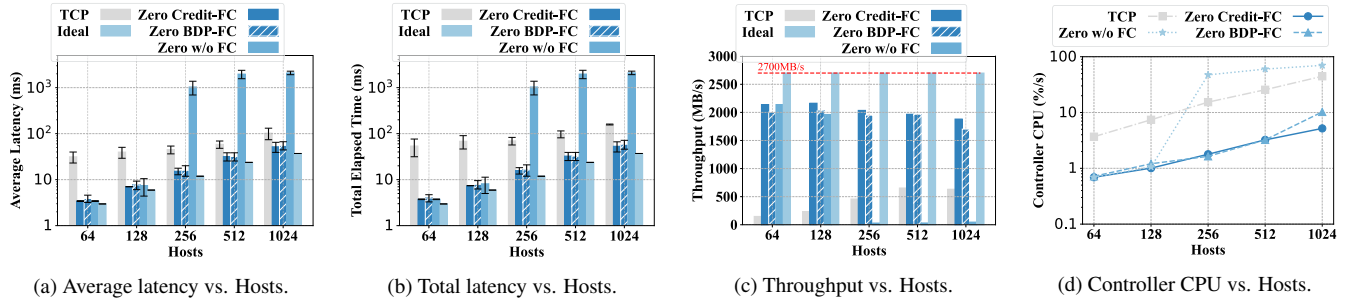


Figure 10: Monitor performance with 64 – 1024 hosts × 128KB data in Cluster1.

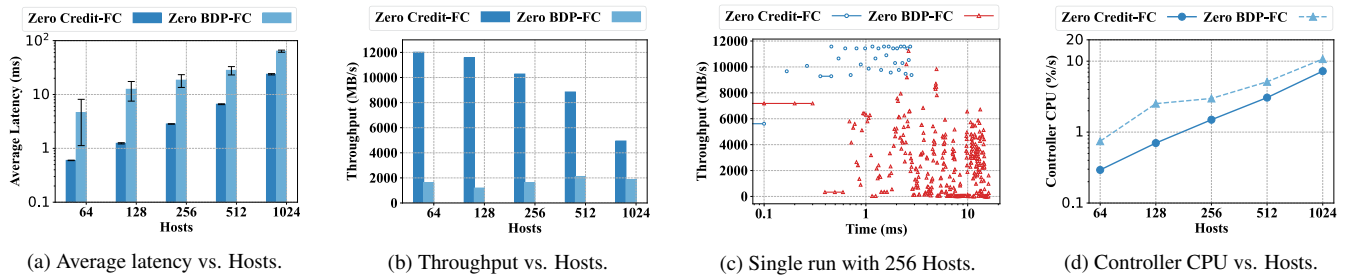


Figure 11: Monitor performance with 64 – 1024 hosts × 128KB data in Cluster2.

With $10\times$ lower sampling interval, the Netdata CPU increases $10\times$, while the gap of CPU utilization between single- and multi-thread becomes larger. However, the multi-thread Netdata achieves $0.7 \sim 1ms$ latency (Figures 7b and 7d). Netdata thus trades off CPU utilization with latency. On the contrary, ZERO has negligible CPU utilization ($< 0.1\%$) in its control plane, which is a one-off expense independent of sampling intervals as shown in Figure 7c. ZERO data plane has zero CPU overhead denoted by the tiny pillar in Figures 7a and 7c. ZERO reduces latency by one/two order of magnitudes compared with Netdata, as shown in Figures 7b and 7d.

Linux kernel case. Figure 8 illustrates the performance of Linux kernel case, where ZERO monitors multiple container instances. As shown in Figures 8a and 8c, the CPU utilization of the ZERO data plane is zero. The CPU utilization of ZERO control plane is high, because of the frequent invocations of `mmap` when registering per-CPU kernel metrics. However, the total CPU time of ZERO control plane is fixed, *i.e.*, $3.4 \sim 14.7$ ms to register $10 \sim 40$ instances. As shown in Fig 8c, the overhead of ZERO control plane is only 0.8% independent of sampling intervals, while the Netdata CPU increase to 4% with $10ms$ sampling interval. ZERO achieves $0.07 \sim 0.1ms$ latency, which is an order of magnitude lower than that of Netdata ($0.5 \sim 1.6ms$). ZERO may have higher READ latency than the uploading latency of Netdata. Because kernel metrics are per-CPU counters, ZERO controller needs n requests to obtain n copies of counters in each CPU.

eBPF case. Figure 9 shows the performance of monitoring TCP connections of with NX, NX-eBPF and ZERO. NX monitor has the highest CPU utilization ($1.6 \sim 59\%$) and latency

($16 \sim 250ms$), due to its outdated implementation, which traverses all TCP connections using the kernel `tcp_diag` interface. NX-eBPF introduces eBPF to low the overhead of getting TCP info, while ZERO further eliminates the syscall and memory copy overhead of reading eBPF map. As shown in Figures 9b and 9d, ZERO performs $40\% \sim 80\%$ lower latency than that of NX. Similar with Redis and Linux kernel case, the CPU utilization of ZERO data plane is still zero.

6.3 ZERO Scalability

We then evaluate the scalability of ZERO. The controller adopts receiver-driven model to obtain raw metrics, which issues TCP-based RPC or READ requests to the agent. The collecting phase at the monitored host is omitted to compare the raw performance. The controller adopts busy polling by default. The ideal result is obtained via `ib_read_bw` [16]. Our three key observations are summarized as follows.

READ achieves better performance. The TCP-based baseline achieves much higher average latency compared with ZERO (Figure 10). The total elapsed time of collecting metrics from all hosts is usually $2 \sim 3\times$ the average latency in TCP, resulting in low throughput. We analyze that TCP suffers from low efficient congestion control (CC) [20, 37] and the processing overhead of kernel stack [24]. Both factors incur large delay to concurrent RPC requests, resulting in variations of starting/ending time. On the contrary, ZERO eliminates the overhead of kernel stack via RDMA. The average latency and elapsed time of ZERO are nearly equivalent (Figures 10a and 10b). ZERO also achieves lower CPU compared with the baseline as shown in Figure 10d.

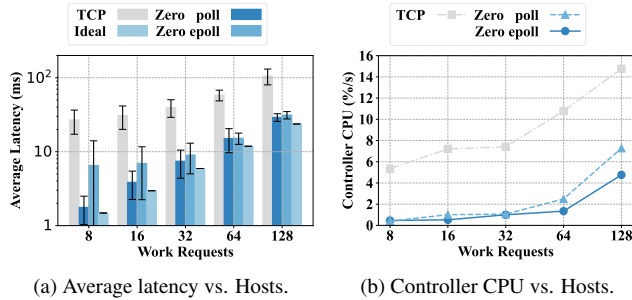


Figure 12: Monitor performance with 8 – 128 WRs × 4KB.

Credit-FC avoids triggering ECN/PFC. In Cluster1, we evaluate the ZERO performance w/ Credit-FC and BDP-FC, and w/o any flow control (FC). For tests w/o FC, the controller posts all requests in the beginning. The BDP-FC adopts a fixed credit of BDP (16KB with 25GbE bandwidth and $5 \sim 6\mu s$ RTT). Interestingly, PFC pause is not triggered until 256 QPs for ZERO w/o FC, due to the large dynamic buffer of switches [17] and the high threshold of PFC pause action (XOFF) [95]. Besides, the RNIC limits the maximum of outstanding READ requests to 16. Accordingly, the total credit is set to $8MB = 128 \times 64KB$. As shown in Figure 10a, both Credit-FC and BDP-FC effectively avoid PFC. The gap between ZERO and ideal is origin from the extra overhead in monitoring (Figure 10c), *e.g.*, virtualization, initializing and posting work requests (WRs) for each VM. The 25GbE bandwidth is still saturated with the $C_f \geq BDP$.

In Cluster2, the network has much higher BDP (96KB with 200GbE bandwidth and $4\mu s$ RTT) and the ECN is enabled by default. We observe a large fraction of ECN marked packets and high latency in BDP-FC (Figure 11a), because the ECN minimum threshold [95] is set to $\sim 1000KB$. Credit-FC still works with 512/1024 QPs due to the posting and arrival delay of READ requests with single thread (Section 4.3), where the build-up buffer should reach $2/4MB$ with the smallest quota of 4KB. We observe similar phenomenon with 1024 QPs in Cluster1. As shown in Figure 11b, Credit-FC only utilizes half of the 200 GbE bandwidth — because the credit is much smaller than the BDP of network to avoid congestion/ECN ($4 \sim 16KB$ vs. 96KB) — the bottleneck lies in the switch buffer/threshold instead of end host. The throughput also degrades rapidly because large number of QPs share RNICs at both agent and controller.

We next zoom into the difference between Cluster1 and Cluster2. Cluster1 set much higher PFC threshold than the ECN threshold in Cluster2, resulting in larger credit to saturate the bandwidth. However, the tradeoff is that the pause duration occupies $\sim 99\%$ (Figure 10b) when PFC is triggered. As a comparison, ECN reacts to congestion and recovers traffic rapidly (Figure 11c). Another benefit of Credit-FC is reducing the CPU utilization for busy polling via avoiding network congestion as shown in Figures 10d and 11d.

Hybrid-I/O model is highly CPU efficient. We then evaluate the hybrid-I/O model in Cluster1. We post all requests in the beginning as the maximum outstanding READ requests is bounded to 16 by RNIC. As shown in Figure 12b, the CPU utilization of both epoll and polling increases with large data size, while epoll achieves lower CPU utilization. To highlight the gap, we set lower sampling interval to monitor 128 hosts with 256KB data. Results show that the CPU utilization of epoll and polling are 13.5/40.9% and 20.2/72.6 respectively with 100/10ms interval. However, the event-driven polling has high performance variations with small number of requests (Figure 12a), due to *epoll* syscall and thread blocking. With large-sized data, such overhead is averaged across many in-flight requests. We thus adopt epoll for general cases ($32 \sim 128$ requests) and polling for hosts with small-sized data (< 32 requests).

7 Experience and Future Work

In this section, we share our experience of building large-scale monitoring system using RDMA.

Achieving high scalability and availability. The scalability of RDMA-based distributed systems is limited by the number of QPs [26, 62, 82], which are cached in the limited SRAM of RNIC. Even several works [46, 47] adopt UD to reduce the number of QPs, ZERO uses READ to bypass the monitored host, which is only supported in RC mode. ZERO adopts QP sharing and grouping to remedy the QP constraints. First, ZERO agent manages all system/application metrics in a host sharing one QP connection. Second, ZERO controller monitors a group of QPs/hosts in each period, *e.g.*, 64 hosts with $1 \sim 10ms$ period, then switches to another group of QPs/hosts to avoid frequent QP evictions. To achieve high availability, each agent will build QP connections with at least three controllers in the practical deployment. Similar to most distributed systems, all these controllers run a consensus-based coordination service [40] to detect failures, and ZERO can switch to a standby controller seamlessly when the active controller is down.

Avoiding network interference. In the practical deployment, monitoring traffics co-exist with service traffics and inevitably impacts the network performance of services, due to the contentions at both agent and switches. Note that the controller has no such concerns with dedicated server for monitoring. Before ZERO, existing deployment adopts several mechanisms for traffic isolation, which inevitably brings other side effects. For example, a thorough solution is physically isolating traffics of services and monitoring with independent NICs and links [2]. However, physical isolation incurs large CapEx and is only suitable to high-priority services necessitating high SLA. Another solution is assigning a separate and lower-priority queue for monitoring traffics [35]. The persistent high loads of services may cause starvation of mon-

itoring traffics and losses of data in consecutive monitoring periods. Besides, ZERO built on RDMA is sensitive to such timeouts, which may cause QP state machine errors [72]. We thus abandon the traditional method of traffic isolation and resort to receiver-driven CC to avoid network interference.

ZERO provides a new perspective to mitigate network interference, *i.e.*, limiting the credit of monitoring traffics when co-existing. Specifically, the controller adopts group switching with 64 QPs/hosts in each group. The QPs of next group will be pre-fetched to RNIC SRAM for warming up. The total credit T in Credit-FC is set to 256KB, which maximally adds $\sim 20/10\mu s$ queuing delay with 100/200GbE bandwidth. Note that the maximum build-up queue is much less than 256KB due to the posting and arrival delay (Section 6.3). As a comparison, the traditional send-driven model easily causes network jitters with burst traffics. Besides, such settings have a negligible impact on monitoring QoS and single controller supports $1 \sim 10k$ hosts with $0.1 \sim 1s$ intervals. The agent only occupies $0.01 \sim 0.1\%$ bandwidth of the monitored host.

Receiver-driven CC. Compared with existing sender-driven CC, the receiver-driven CC achieves several benefits in monitoring. Existing CC mechanisms, *e.g.*, DCQCN [95] and TIMELY [69], react to congestion after switch buffer/queue reach threshold. Besides, they aim to achieve equal bandwidth sharing across multiple flows, and cannot avoid interference between service traffics and monitoring traffics. As a comparison, the receiver-driven CC avoids network congestion and interference in advance by limiting the total in-flight data of monitoring. On the other hand, existing CC mechanisms, *e.g.*, DCQCN and HPCC [57], are complex in deployment and requires ECN or in-network telemetry (INT) supports from switches. However, ECN or INT capability are not always supported, *e.g.*, in Cluster1. The Credit-FC in our deployment is simple and effective to avoid triggering PFC/ECN.

We also observe several limitations of current Credit-FC. It only adopts credit without pacing [50] and cannot support massive concurrent flows with a 4KB transmission fragment. Besides, it is not a universal CC mechanism for data-center traffics, which dedicates to avoid network congestion and interference caused by monitoring traffics in the ZERO framework. In our future work, we will try to resolve these limitations from two aspects. First, we will consider both host bandwidth and ECN threshold [88] with a combination of credit- and pacing-based CC, to achieve full bandwidth utilization while avoiding network congestion. Second, we will explore the universal receiver-driven model in cloud networks, which has the benefits of CPU offloading via RDMA and more convenient CC [77].

8 Related Work

One-sided RDMA. In the system area, it is a trend to leverage one-sided RDMA operations to bypass the server CPUs.

As pioneering works, Pilaf [68] enables the clients to directly read data from the server memory via RDMA read. Clients use CRC64 to search for the inconsistency of data caused by the possible read-write races on the server. RFP [79, 86] explores one-sided RDMA to provide another alternative solution for RPC, which uses RDMA read to fetch the response result. On the other hand, several works [46, 61, 86] explore how to optimize the raw performance of one-sided RDMA. To the best of our knowledge, ZERO is the first work to leverage one-sided RDMA for distributed monitoring.

Monitoring system. There are plenty of works targeting for the design of monitoring system [65]. Yet, all these works focus on data analytic [51, 80], tracing bugs [59, 63] and visualization [71]. Distinct from these works, where monitors are tightly coupled with the monitored applications and hosts, ZERO decouples the monitor from the monitored infrastructure and eliminates the monitoring overhead completely.

Cloud-native monitoring. Netdata [10] enables users to quickly identify and troubleshoot issues, and make data-driven decisions according to the pre-built visible dashboards. Prometheus [11] is an open-sourced monitoring system with complete ecosystem to extract time series data from the cloud-native applications, and it focuses on collecting metrics via a powerful query language called PromQL. Stackdriver [4] is the logging and monitoring solution of Google, which is integrated tightly into Google Cloud. Likewise, ZERO is deeply used in the cloud-native ecosystem of Alibaba cloud.

9 Conclusion

We propose the ZERO monitoring system framework, exploiting one-sided RDMA read for remote monitoring. ZERO achieves zero-overhead monitoring via the novel control plane and data plane. ZERO supports large-scale distributed monitoring via credit-based FC and hybrid I/O model. ZERO thus paves the way for integrating RDMA into the monitoring systems, which desires to benefit from the high performance of RDMA while avoiding poor scalability. We deploy ZERO in Alibaba cloud-native platform to evaluate its performance. The deployment results show that ZERO resolves interference problem of traditional monitor and easily fulfills both latency and throughput requirements in cloud-native monitoring.

10 Acknowledgment

We sincerely thank the anonymous shepherd and reviewers for their insightful comments and feedback. This work was supported in part by NSFC grant 62141220, 61972253, U1908212, 72061127001, 62172276, 61972254, the Program for Professor of Special Appointment (Eastern Scholar) at Shanghai Institutions of Higher Learning, Alibaba Innovative Research (AIR) Program. Corresponding author: Linghe Kong (linghe.kong@sjtu.edu.cn).

References

- [1] Alibaba Cloud Container Service for Kubernetes. <https://www.alibabacloud.com/en/product/kubernetes>, Dec. 2020.
- [2] Best Practices of ECS Container Network Multi-NIC Solution. <https://www.alibabacloud.com/blog/593997>, Dec. 2020.
- [3] Extended Berkeley Packet Filter. <https://ebpf.io/>, Dec. 2020.
- [4] Google cloud's operations suite (formerly stackdriver). <https://cloud.google.com/products/operations>, Dec. 2020.
- [5] Grafana. <https://grafana.com/>, Dec. 2020.
- [6] Hubble. <https://github.com/cilium/hubble>, Dec. 2020.
- [7] InfluxDB. <https://www.influxdata.com/>, Dec. 2020.
- [8] Install netdata on cloud providers. <https://learn.netdata.cloud/docs/agent/packaging/installer/methods/cloud-providers>, Dec. 2020.
- [9] MaxCompute - Conduct large-scale data warehousing with MaxCompute. <https://www.alibabacloud.com/product/maxcompute>, Dec. 2020.
- [10] Netdata - Monitor everything in real time for free with Netdata. <http://www.netdata.cloud>, Dec. 2020.
- [11] Prometheus - Monitoring system & time series database. <https://prometheus.io/>, Dec. 2020.
- [12] Redis. <https://redis.io/>, Dec. 2020.
- [13] Specifying Attributes of Variables. <https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Variable-Attributes.html>, Dec. 2020.
- [14] ApsaraDB for Redis. <https://www.alibabacloud.com/product/apsaradb-for-redis>, June. 2021.
- [15] Cloud-native applications management. <https://www.alibabacloud.com/en/solutions/container>, June. 2021.
- [16] OFED performance test suite. <https://github.com/linux-rdma/perftest>, June. 2021.
- [17] Packet buffer of switches. <https://people.ucsc.edu/~warner/buffer.html>, June. 2021.
- [18] Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.
- [19] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *USENIX ATC*, 2018.
- [20] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM*, 2010.
- [21] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *USENIX ATC*, 2018.
- [22] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *USENIX ATC*, 2019.
- [23] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with RDMA and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.
- [24] Qizhe Cai, Shubham Chaudhary, Midhul Vuppapapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *ACM SIGCOMM*, 2021.
- [25] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, 2019.
- [26] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *EUROSYS*, 2019.
- [27] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *SOSP*, 2017.
- [28] Christina Delimitrou and Christos Kozyrakis. Hcloud: Resource-efficient provisioning in shared cloud systems. In *ASPLOS*, 2016.
- [29] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. Detecting asymmetric application-layer denial-of-service attacks in-flight with finelame. In *USENIX ATC*, 2019.
- [30] Luca Deri, Samuele Sabella, and Simone Mainardi. Combining system visibility and security using eBPF. In *ITASEC*, 2019.

- [31] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *USENIX NSDI*, 2014.
- [32] Yihui Feng, Zhi Liu, Yunjian Zhao, Tatiana Jin, Yidi Wu, Yang Zhang, James Cheng, Chao Li, and Tao Guan. Scaling large production clusters with partitioned synchronization. In *USENIX ATC*, 2021.
- [33] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems. In *ASPLOS*, 2019.
- [34] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *ASPLOS*, 2019.
- [35] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *ACM SIGCOMM*, 2016.
- [36] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *IEEE/ACM IWQoS*, 2019.
- [37] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.
- [38] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress data path: Fast programmable packet processing in the operating system kernel. In *ACM CONEXT*, 2018.
- [39] Michio Honda, Felipe Huici, Costin Raiciu, Joao Araujo, and Luigi Rizzo. Rekindling network protocol innovation with user-level stacks. *ACM SIGCOMM Computer Communication Review*, 44(2):52–58, 2014.
- [40] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, volume 8, 2010.
- [41] Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments. In *ACM SOCC*, 2019.
- [42] Congfeng Jiang, Yitao Qiu, Weisong Shi, Zhefeng Ge, Jiwei Wang, Shenglei Chen, Christophe Cerin, Zujie Ren, Guoyao Xu, and Jiangbin Lin. Characterizing co-located workloads in Alibaba cloud datacenters. *IEEE Transactions on Cloud Computing*, 2020.
- [43] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance modeling for cloud microservice applications. In *ACM/SPEC ICPE*, 2019.
- [44] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *USENIX NSDI*, 2019.
- [45] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM*, 2014.
- [46] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance RDMA systems. In *USENIX ATC*, 2016.
- [47] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided RDMA datagram RPCs. In *USENIX OSDI*, 2016.
- [48] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating network-based {CPU} in container environments. In *USENIX NSDI*, 2018.
- [49] Ricardo Koller and Dan Williams. Will serverless end the dominance of linux in the cloud? In *ACM HOSTOS*, 2017.
- [50] Gautam Kumar, Nandita Dukkupati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *ACM SIGCOMM*, 2020.
- [51] Mahendra Kutare, Greg Eisenhauer, Chengwei Wang, Karsten Schwan, Vanish Talwar, and Matthew Wolf. Monalytics: online monitoring and analytics for managing large scale data centers. In *International Conference on Autonomic Computing*, 2010.
- [52] Sándor Laki, Gergő Gombos, Péter Hudoba, Szilveszter Nádas, Zoltán Kiss, Gergely Pongrácz, and Csaba Keszei. Scalable per subscriber QoS with core-stateless scheduling. *ACM SIGCOMM Demo*, 1:84–86, 2018.
- [53] Sándor Laki, Dániel Horpácsi, Péter Vörös, Róbert Kitlei, Dániel Leskó, and Máté Tejfel. High speed packet forwarding compiled from protocol independent data plane specifications. In *ACM SIGCOMM*, 2016.

- [54] Joshua Levin. Viperprobe: Using eBPF metrics to improve microservice observability, 2020.
- [55] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *ACM SIGCOMM*, 2019.
- [56] Feifei Li. Cloud-native database systems at Alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment*, 12(12):2263–2272, 2019.
- [57] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. HPC: High precision congestion control. In *ACM SIGCOMM*, 2019.
- [58] Qixiao Liu and Zhibin Yu. The elasticity and plasticity in semi-containerized co-locating cloud workload: a view from alibaba trace. In *ACM SOCC*, 2018.
- [59] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *USENIX NSDI*, 2020.
- [60] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *ACM SoCC*, 2021.
- [61] Teng Ma, Kang Chen, Shaonan Ma, Zhuo Song, and Yongwei Wu. Thinking more about RDMA memory semantics. In *IEEE International Conference on Cluster Computing (CLUSTER)*, 2021.
- [62] Teng Ma, Tao Ma, Zhuo Song, Jingxuan Li, Huaixin Chang, Kang Chen, Hai Jiang, and Yongwei Wu. X-RDMA: Effective RDMA middleware in large-scale production environments. In *IEEE International Conference on Cluster Computing (CLUSTER)*, 2019.
- [63] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *SOSP*, 2015.
- [64] Ilias Marinos, Robert NM Watson, and Mark Handley. Network stack specialization for performance. *ACM SIGCOMM Computer Communication Review*, 44(4):175–186, 2014.
- [65] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.
- [66] Steven McCanne and Van Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *USENIX winter*, 1993.
- [67] Bradley Miles and Dave Cliff. A cloud-native globally distributed financial exchange simulator for studying real-world trading-latency issues at planetary scale. *arXiv preprint arXiv:1909.12926*, 2019.
- [68] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *USENIX ATC*, 2013.
- [69] Radhika Mittal, Vinh The Lam, Nandita Dukkupati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *ACM SIGCOMM*, 2015.
- [70] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *ACM SIGCOMM*, 2018.
- [71] Alex Page, Tolga Soyata, Jean-Philippe Couderc, Mehmet Aktas, Burak Kantarci, and Silvana Andreescu. Visualization of health monitoring data acquired from distributed sensors for multiple patients. In *IEEE GLOBECOM*, 2015.
- [72] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. {RDMA} is turing complete, we just did not know it yet! In *USENIX NSDI*, 2022.
- [73] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *5th Annual Linux Showcase & Conference*, 2001.
- [74] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the cost of atomic operations on modern architectures. In *IEEE PACT*, 2015.
- [75] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed os for hardware resource disaggregation. In *USENIX OSDI*, 2018.
- [76] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *ASPLOS*, 2019.
- [77] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In *ACM SIGCOMM*, 2020.
- [78] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. Softsku: optimizing server architectures for microservice diversity@ scale. In *ISCA*, 2019.

- [79] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is faster than server-bypass with RDMA. In *EUROSYS*, 2017.
- [80] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. Sieve: actionable insights from monitored metrics in distributed systems. In *ACM/IFIP/USENIX Middleware Conference*, 2017.
- [81] Linus Torvalds. The linux edge. *Communications of the ACM*, 42(4):38–39, 1999.
- [82] Shin-Yeh Tsai and Yiyang Zhang. Lite kernel RDMA support for datacenter applications. In *SOSP*, 2017.
- [83] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications. *ACM Computing Surveys*, 53(1):1–36, 2020.
- [84] Hanzhang Wang, Phuong Nguyen, Jun Li, Selcuk Koprulu, Gene Zhang, Sanjeev Katariya, and Sami Ben-Romdhane. Grano: Interactive graph-based root cause analysis for cloud-native distributed data platform. *Proceedings of the VLDB Endowment*, 12(12):1942–1945, 2019.
- [85] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. ICTCP: Incast congestion control for TCP in data-center networks. *IEEE/ACM transactions on networking*, 21(2):345–358, 2012.
- [86] Yongwei Wu, Teng Ma, Maomeng Su, Mingxing Zhang, Kang Chen, and Zhenyu Guo. RF-RPC: Remote fetching RPC paradigm for RDMA-enabled network. *IEEE Transactions on Parallel and Distributed Systems*, 30(7):1657–1671, 2018.
- [87] Xin Xu and Bhavesh Davda. A hypervisor approach to enable live migration with passthrough sr-iov network devices. *ACM SIGOPS Operating Systems Review*, 51(1):15–23, 2017.
- [88] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. ACC: Automatic ECN tuning for high-speed datacenter networks. In *ACM SIGCOMM*, 2021.
- [89] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Filemr: Rethinking RDMA networking for scalable persistent memory. In *USENIX NSDI*, 2020.
- [90] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-latency networking with the OS stack and dedicated NICs. In *USENIX NSDI*, 2016.
- [91] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, et al. Analyticdb: Real-time olap database system at alibaba cloud. *Proceedings of the VLDB Endowment*, 12(12):2059–2070, 2019.
- [92] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. High-density multi-tenant bare-metal cloud. In *ASPLOS*, 2020.
- [93] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In *Proceedings of the VLDB Endowment*, volume 7, pages 1393–1404, 2014.
- [94] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling wechat microservices. In *ACM SOCC*, 2018.
- [95] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *ACM SIGCOMM*, 2015.

CRISP: Critical Path Analysis of Large-Scale Microservice Architectures

Zhizhou Zhang
University of California, Santa Barbara

Murali Krishna Ramanathan
Uber Technologies

Prithvi Raj
Uber Technologies

Abhishek Parwal
Uber Technologies

Timothy Sherwood
University of California, Santa Barbara

Milind Chabbi
Uber Technologies

Abstract

Microservice architectures have become the lifeblood of modern service-oriented software systems. Remote Procedure Calls (RPCs) among microservices are deeply nested, asynchronous, and large in number, thus making it very hard to identify the underlying service(s) that contribute to the overall end-to-end latency experienced by a top-level request. State-of-the-art RPC tracing tools collect a deluge of data but provide little insight. We need sophisticated tools to bubble-up signals from a myriad of RPC traces to assist developers in identifying optimization opportunities, pinpointing common bottlenecks, setting appropriate time outs, diagnosing error conditions, and planning and managing compute capacity, to name a few.

In this paper, we present CRISP — a tool to perform critical path analysis (CPA) over a large number of traces collected from RPCs in microservices environments. CRISP provides three critical performance analysis capabilities: a) a *top-down* CPA of any specific endpoint, which is tailored for service owners to drill down the root causes of latency issues, b) a *bottom-up* CPA over all endpoints in the system — tailored for infrastructure and performance engineers — to bubble up those (interior) APIs that have a high impact across many endpoints, and c) an on-the-fly anomaly detection to alert potential problems.

We have applied CRISP’s capabilities on Uber’s entire backend system composed of $\sim 40\text{K}$ endpoints that cater to real-time requests from more than 100 million active daily users worldwide. Using the critical path as the basis of performance analysis has a) helped us identify five performance issues and optimization opportunities across two business-critical microservices, b) guided us in our future hardware choice that reduces end-to-end latencies, and c) reduced the false positives in anomaly detection by up to 50% while speeding up the training and inference by up to $28\times$ and up to $67\times$, respectively, over the state of the art.

1 Introduction

Microservice architectures [23, 27, 28, 36, 43, 45, 56] have become the lifeblood of modern service-oriented software sys-

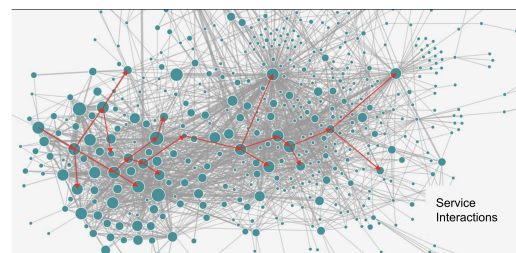


Figure 1: Complex microservice RPC call graph at Uber collected via Jaeger tracing.

tems. As opposed to monolithic software development and deployment, in a microservice environment, the business logic is broken into individually deployable programs, which allow fast development and scalable deployment. Individual microservice *instances* interact with one another via remote procedure calls (RPCs). As microservices evolve with the business, they grow in number and their interactions become complex.

Uber’s backend is an exemplar of microservice architecture. Uber has $\sim 4,000$ microservices interacting with each other via RPCs. Each microservice hosts a handful of APIs, leading to a total of about 40,000 unique RPC endpoints that can call one another in complex ways, as depicted in Figure 1. Hereafter, we use the terms *endpoint* and *API* interchangeably to mean a uniquely named functionality provided by a service. We use the terms *operation* and *RPC* interchangeably to mean an instance of invocation of such an API.

A service request arriving at an entry point API to the Uber backend systems undergoes multiple “hops” through numerous microservice RPCs before being fully serviced. The life of a request results in intricate microservice interactions. These interactions are deeply nested, asynchronous, and invoke numerous other downstream APIs. As a result of this complexity, it is very hard to identify which underlying service(s) contribute to the overall end-to-end latency [21, 32, 38, 44, 52, 53, 63] experienced by a top-level request. Answering this question is critical in many situations. For example:

- Identifying optimization opportunities for a top-level

- microservice (e.g., reducing tail latency)
- Identifying bottleneck APIs that affect numerous endpoints
- Setting appropriate time-to-live values for RPCs
- Diagnosing outages and error conditions
- Planning for computing and other capacity management

The critical path [59] is the longest chain of dependent tasks in a microservice dependency graph. Reducing the critical path length is necessary to reduce the end-to-end latency of a request. Hence, latency optimization efforts benefit from prioritizing the services that are on the critical path.

We have developed a tool, CRISP¹, to pinpoint and quantify performance problems in microservice architectures. CRISP uses the RPC tracing facility provided by Jaeger [6] and constructs the critical path through a request's graph of dependencies. The critical path may vary among requests; hence, CRISP computes the critical path per request. It then aggregates and summarizes critical paths from millions of requests. Finally, it presents them as digestible and actionable insights via rich heat maps [5] and flame graphs [31]. CRISP provides knobs to dissect the details with different percentile values that help in performance diagnoses.

As a full-fledged performance analysis tool, CRISP caters to various use cases via the following rich set of capabilities that scale to work on millions of traces:

- **Top-down analysis:** A top-down analysis of any specific endpoint of interest. This capability allows service owners to deep dive into their specific endpoint and pinpoints and quantifies bottlenecks encountered in the RPC dependency graph. Improving these bottlenecks should be the first-order priority to reduce the latency of the endpoint.
- **Bottom-up analysis:** A bottom-up analysis over all endpoints, which bubbles up and ranks by the impact of those interior APIs that cause the most latency across most endpoints. Optimizing these interior APIs reduces latency across numerous endpoints.
- **Neural network-based anomaly detection:** An automated anomaly detection system, which detects whether a request is exhibiting abnormal behavior compared with the past history of the endpoint. The system is trained per endpoint using an autoencoder-decoder machine learning technique [39]. This system is set up to expedite problem detection and alert developers. Basing the abnormality detection on the divergence in the critical path as opposed to the full call graph [39] not only makes the training and inference faster but also reduces false alerts.

Practical deployment of CRISP at Uber over a three-month period working on 40K endpoints while processing $\sim 200GB$ of traces with ~ 18 million spans in ~ 256 hours of CPU time per day has resulted in the following impact:

- Detection and narrowing down the causes of five latency impacting bugs in two business-critical services
- Identification of a $1.5\times$ tail latency lengthening due to

hardware choice and the resulting guidance for future hardware selection

- Up to $27.77\times$ speedup in training, up to $66.85\times$ speed up in inference, and 50% reduction in false alerts in identifying abnormality of service behaviors over the state of the art [39]

The rest of this paper is organized as follows: Section 2 motivates CRISP with a use case at Uber, Section 3 describes the Jaeger tracing framework, Section 4-6 describe the methodology, internals, and features of CRISP, Section 7 evaluates CRISP at Uber, Section 8 discusses the related work, and Section 9 offers our conclusions.

2 Motivating Example for CRISP

Fulfillment [8] at Uber is a platform to orchestrate and manage the lifecycle of orders and user sessions with millions of active participants. The Fulfillment platform is a foundational Uber capability that enables the rapid scaling of new verticals. The platform handles more than a million concurrent users and billions of trips per year that span over ten thousand cities. The platform handles billions of database transactions a day. Hundreds of Uber microservices rely on the platform as the source of truth for the accurate state of the trips and driver or delivery sessions. Events generated by the platform are used to build hundreds of offline datasets to make critical business decisions. Over 500 developers extend the platform using APIs, events, and code to build more than 120 unique fulfillment flows.

The `createOrder` endpoint allows capturing the requester's intent in the Uber backend. Intent can be to request a ride from one of the ridesharing lines of products, food booked and dispatched by one of the courier partners, or a package be delivered to a customer. This endpoint has a complex task dependency graph necessary for: a) determining order risk such as user fraud, sufficient user balance via authentication hold, b) ensuring the fare presented to the requester in the shopping phase is still valid, c) determining the benefits the requester is eligible for, d) enriching data with location information, and e) creating an order in the backend to start the matching process.

The tasks in this endpoint have grown organically as requirements evolved. This has led to an increase in p95 latency to 6 seconds, affecting user experience. The service itself is written in Java, and highly (both macro and micro) optimized using periodic profiling. However, the profiling offered no insights into downstream calls, where most time is spent. Quantitative insight into the causes of the latency was hard to analyze by looking at individual traces because each trace contains thousands of nested and overlapping RPCs.

There are numerous sampling- and instrumentation-based profilers [10, 11, 13, 29] for intra-service profiling. However, they do not collect metrics at the individual request level. The Fulfillment microservice (as most other microservices) is highly threaded; the work of an individual request may be partitioned among multiple threads within a process as well

¹named taking letters from `critical` and `span`

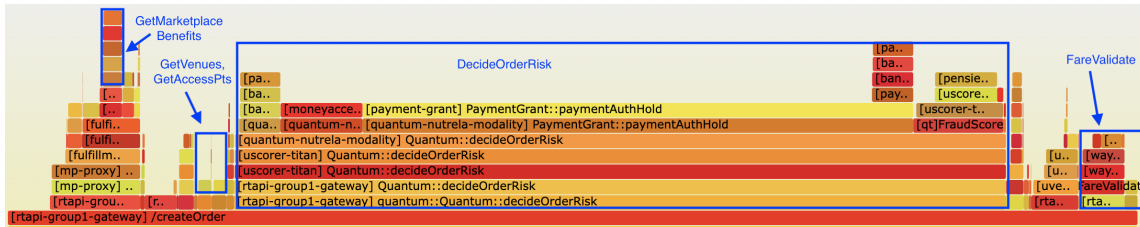


Figure 2: Critical path(s) of createOrder endpoint shown as a flame graph via CRISP after processing 100K Jaeger traces.

as multiple threads may be handling independent requests simultaneously. In such a setup, traditional profilers fail to highlight the causes of latencies incurred at an individual request level. Also, traditional profilers fail to capture IO waiting, task dependencies, and serialization patterns.

With CRISP, the development team performed a top-down critical path analysis of this endpoint over 100K traces (~200GB of traces) and visualized the results as a flame graph as shown in Figure 2. Navigating the “hot” critical paths via the flame graph not only corroborated an existing hunch while offering quantitative guidance but also shed light on new optimization opportunities lurking in the wild. Below, we enumerate a few defects and optimization opportunities that became evident by inspecting CRISP-provided insights.

Async flow optimization: `decideOrderRisk` contributes to about 68% of the end-to-end P50 latency, revealing the following optimizations: a) aggressively use cache in `FraudScore` to reduce its latency and b) parallelize the calls beneath this big endpoint (e.g., `PaymentAuthHold` and `FraudScore`). In the long term, the team envisions using an asynchronous invocation of `paymentAuthHold` and using notification to the requester when a provider is assigned.

Unnecessary API serialization: There was an unnecessary serialization between `GetVenues` and `GetAccessPts`. These two RPCs can be done in parallel.

Avoidable server roundtrip for validation: `FareValidate` contributes to about 5% of the end-to-end P50 latency. This is a call that need not be performed every time. Trusted edge devices (e.g., company mobile app) can validate at the edge improving performance for trusted users and falling back to server validation if the fare has expired based on fare expiry TTL; untrusted apps will use the full server validation.

Caching over DB fetch: `GetMarketplaceBenefits` contributes to about 5% of the P50 latency. This can be served via a cache rather than a database read to fetch requester benefits.

3 Background

In this section, we first describe the microservice tracing infrastructure at Uber and then enumerate its shortcomings.

3.1 Distributed Tracing at Uber

Microservices run over several physical hosts, usually owned by multiple teams, and written in multiple languages. It is impossible to use traditional profilers [7, 13, 29] to gain insight into the events involved in processing a request. Because each physical host can have a separate clock, it is intractable to infer causality using time alone. Distributed tracing [47] encodes causality information in a distributed context, which is propagated across process boundaries. It provides a way to infer states across various services for the lifetime of a request.

At Uber, Jaeger [6] is used as the distributed tracing system. Jaeger provides clients for generating trace data and components for storage and retrieval of traces. Microservices instrumented with Jaeger clients produce OpenTracing [6]-compliant spans when receiving new requests and attach distributed context information (trace ID, span ID, custom key-value pairs). The “span” [46] is the primary building block of a distributed trace, which represents a serial unit of work done in a distributed system. Each span contains the following information:

- API name
- Start and finish timestamps
- Custom key-value pairs
- Span context and references (described below)

Each span may reference other spans with a causal relationship by span context. A span may reference a parent with the `ChildOf` relationship, indicating that the parent span waits for the child to finish a certain task. Multiple child spans can be referenced by the same parent and run concurrently.

While the source code is always instrumented, the overhead is controlled by a dynamic sampling rate, which is adjusted based on the traffic received by Internet-facing endpoints. No data is collected for traces that are not sampled. Specifically, adaptive sampling sets a target QPS for traces on a per root service-endpoint basis, which ensures that the number of samples on the external API request remains roughly constant. Jaeger does not support tail-based sampling [12].

Figure 3 depicts the Jaeger deployment at Uber. Jaeger is deployed as multiple components, with a `jaeger-agent` running on every host. All applications running on this host send spans to `jaeger-agent` over UDP [9]. `jaeger-agent` then forwards these spans to a `jaeger-collector`, which then buffers spans onto the Kafka [37] distributed event

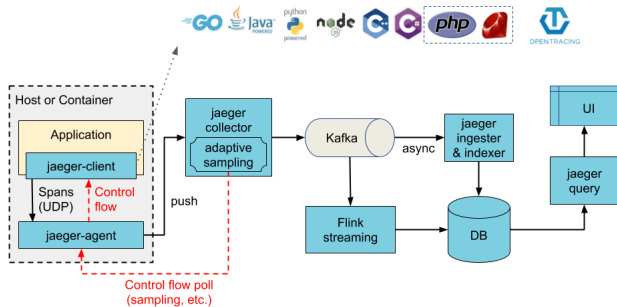


Figure 3: Jaeger deployment at Uber.

streaming platform. The spans buffered in Kafka have multiple consumers: `jaeger-ingester`, which inserts them into Docstore [3], a distributed SQL database, and allows for retrieving full traces; `jaeger-indexer`, which inserts them into Sawmill [4], a schema-agnostic logging platform that allows user-friendly search on spans fields. Additionally, spans are consumed by Apache Flink [15] jobs to produce multi-hop dependency graphs. Depending on the sampling configuration in effect, the backend processes around 400K-1M spans per second, which is approximately 20TB each day. Variance is common due to diurnal patterns.

3.2 Difficulties with Large-Scale Jaeger Traces

Despite their power, Jaeger traces are highly complicated. Jaeger provides a UI to filter traces by time ranges and also provides a UI to view the trace as a callgraph, as well as an expandable tree over a timeline. In spite of these facilities, the users of this manual workflow often complained about the following limitations to analyze endpoint latencies:

- Only first-level insights are possible from drilling down into microservice latencies and errors.
- Using a few Jaeger traces is insufficient to reach a reliable conclusion. Users can visualize and navigate only one Jaeger trace at a time. There is no aggregate summary of traces.
- A single Jaeger trace can be so complex that it is not humanly possible to browse and understand the details. Endpoints commonly have thousands of nodes in the RPC graph with 25-deep call chains and up to 40 spans overlapping in time. It is cumbersome to manually understand the critical path due to the asynchronous nature of calls.
- There is a lack of regular, performance-driven feedback tooling to optimize the workflow or downstream systems.

These challenges introduce a barrier to our developers in effectively using Jaeger to either detect anomalous situations or identify optimization opportunities.

4 CRISP Methodology

The fundamental difficulty in making sense out of a Jaeger trace is due to the complexity of the graph. Our premise is that while the whole graph is interesting in terms of data richness, it brings a lot of noise. There are many RPCs and call paths that are insignificant for a high-level analysis and optimization task. With this understanding, we shrink the graph to its quintessential element—the *critical path*—and aggregate many traces into a single summary that is still rich with call path information.

Critical Path Analysis [25, 59] (CPA) is a well-studied concept over directed acyclic graphs (DAG) formed out of computing graphs in parallel computing. The nodes in the DAG represent tasks (units of serial execution) and the edges represent dependencies between tasks. A node with an out-degree greater than one “spawns” children’s tasks and a node with an in-degree greater than one waits (“syncs”) for the children to finish. Total work is the sum of weights of all nodes and the critical path is the longest weighted path in the DAG.

Definition 1 (Critical Path). In a task graph $G = (V, E)$ made of task vertices V and their dependency edges E , with two special vertices S (start node) and F (finish node), the critical path is a maximal-weight path from S to F . G may contain more than one critical path.

The critical path identifies the sequence of dependent computations that consume the most time. To speed up the service, it is strictly necessary to boost the components on the critical path.

RPCs among microservice operations have a parent-child hierarchical relationship and can be construed as a parallel computation DAG. The deriving critical path from Jaeger traces, however, has the following challenges:

- Unlike a traditional parallel computing DAG seen in the academic literature, the Jaeger traces do not provide clear “spawn” and “sync” events in the DAG.
- The parent spans in Jaeger traces carry no dependence information and so the information of the last “sync” child span is not directly available.
- In order to obtain the last “sync” child span, clock information is needed. However, the clocks on different machines where spans are collected are not time-synchronized.
- The critical path across all requests may not be unique. Services have diurnal patterns and different traces may exhibit different critical paths, which need to be aggregated, and yet “hot” critical paths need to be bubbled up.
- Since the service codes keep evolving, the critical path keeps changing.

We address these challenges in the next section.

We also mention in passing that the CPA is not a performance analysis panacea. Once the exposed latency on the critical path is eliminated, a new critical path may emerge which necessitates the need for an iterative profiling and optimization approach.

5 Critical Path Analysis

In this section, we detail how we compute, aggregate, and represent critical paths from many Jaeger traces for a given endpoint.

5.1 Deriving Critical Path from a Single Trace

CRISP’s trace analysis exploits a map-reduce paradigm to process millions of traces belonging to each endpoint. To this end, each process loads an input Jaeger trace file (JSON format) and builds an n-ary tree, where each parent node is the RPC caller and the children nodes are the immediate downstream callees.

In order to compute the critical path through the trace, we need a computational DAG. To accomplish this under Jaeger/Opentrace trace format, we make use of the start and end times of children’s spans. The start time in every immediate child creates a “spawn” event in the parent and splits its span at that point in time. Similarly, the end time in every immediate child creates a “sync” event in the parent and splits its span at the point in time. Thus, we transform the tree into a logical DAG for critical path construction.

Figure 4 shows an example DAG constructed from Jaeger traces by looking at span start and end times. In Figure 4, the span *A* is the root span, which invokes spans *B*, *X*, and *D*. The span *B* in turn invokes span *C*. The start time T_1 and its end-time T_6 of *B* create a spawn and sync points on *A*, respectively. Similarly, the spans *X* and *D*, create further segments in *A*. Similarly, *B*’s child *C*, creates the spawn and sync points on *B* at T_3 and T_4 , respectively.

Limitations of Jaeger/Opentrace format: One key limitation of the Jaeger is that the parent spans (a.k.a., caller) do not contain dependence information. Specifically, they lack the information of both start and end of callee RPC. Instead, it is the callee that stores both the ID of its parent and callee’s start and end time (per callee’s local clock) in its own span. The implication of the constraint is that the dependency relationship needs to be inferred via clock information recorded in the callee span.

In addition, the inference can be inaccurate because of the clock skew that will be discussed in Section 5.2. Traditionally, the computation of the critical path depends on the last returned child of the parent spans [24]. In Jaeger traces, the last arriving child information is not directly recorded in the parent span. Instead, the last arriver needs to be inferred using the span end time for each child, which will be based on each child’s local clock. Without correctly handling the clock skew, the critical path analysis can go wrong.

One may extend Jaeger tracing by making the callee return additional data to the caller. Unfortunately, ensuring that these changes are adopted universally across thousands of services is an engineering hurdle. Such changes also require support from different RPC libraries used by our system. Our solution, in contrast, does not require such large-scale system-wide

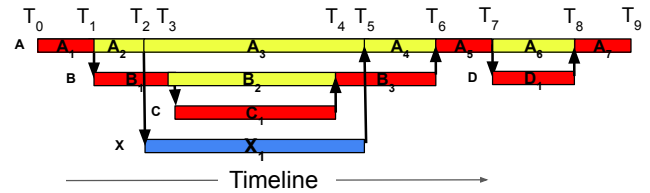


Figure 4: Trace with root span *A*, its children *B*, *X*, and *D*. *B* further calls *C*. CRISP further segments each parent traces based on the start and end time of its children. The red-colored blocks represent the critical path through the trace.

```
def CP(root):
    path = [root]
    if len(root.child) == 0:
        return path
    children = sortDescendingByEndTime(root.children)
    lfc = children[0]
    path.extend(CP(lfc))
    for c in children[1:]:
        if happensBefore(c, lfc):
            path.extend(CP(c))
            lfc = c
    return path
```

Listing 1: Pseudocode to compute critical path.

changes but yet produces high quality results as we describe in the rest of this section.

5.1.1 Critical Path Algorithm

We, first, describe how we compute the critical path in a trace assuming perfectly synchronized clocks in this subsection. We expand to handle unsynchronized clocks in Section 5.2.

The process of computing the critical path (*CP* shown in Listing 1) on the logical DAG starts at the root node *R*—the endpoint under study. We sort all its children by their span *end time* and pick the last finishing child (LFC). The entirety of LFC is on the critical path. Let LFC_s be the start time of the LFC; we ignore all children spans of *R* that may start or end in the time intervening between the start and the end of LFC. We now look for the next child of *R* whose end time immediately precedes LFC_s and perform the same procedure iteratively until no child is left to process. Time not attributed to any child of *R* is attributed to the root span itself.

The process is also recursive. Once an LFC is identified, it recursively calls *CP* on its own children to distribute its time under its children. The result of the *CP* algorithm is a sequence of graph nodes with time associated with each one of them. Applying this algorithm to the trace shown in Figure 4, the critical path is represented by the fragments $A_1B_1C_1B_3A_5D_1A_7$.

There are two types of metrics associated with each node of the critical path — inclusive time and exclusive time. The “exclusive” time does not include the time spent in a node’s callees. The “inclusive” time is the total wall clock time from the start to the end of the RPC on the specific node.

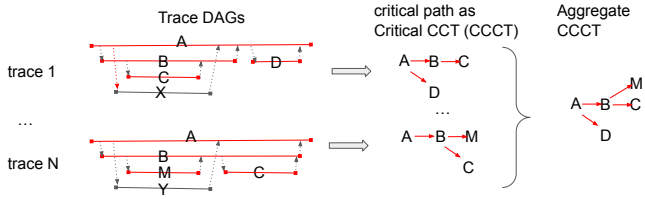


Figure 5: From trace to DAG to critical path (CCCT) to aggregate critical calling context tree. In the trace DAGs (left of the diagram) the x-axis is the flow of time. Horizontal lines are Jaeger spans and vertical lines are caller-callee relationships. Red-colored horizontal spans are on the critical path.

Since every node on the critical path encodes the information on how it was called, and since all call paths originate from a common root — the endpoint under investigation — it enables us to merge all call paths into a calling context tree (CCT) [14] by looking at their common prefixes. Consider the critical path $A_1B_1C_1B_3A_5D_1A_7$ for the trace in Figure 4. This path encodes the following call and return information: A calls B calls C returns to B returns to A calls D returns to A . With this, we can infer that there are the following call chains involved on the critical path: A , $A \rightarrow B$, $A \rightarrow B \rightarrow C$, $A \rightarrow B$, A , $A \rightarrow D$, and A . We can merge all these call paths into a CCT and call it a Critical Calling Context Tree (CCCT). This process is presented in the center section of Figure 5.

The calling context information makes it not only rich but also helps in aggregating critical paths from multiple traces described later in Section 5.3. A level of aggregation happens immediately within each trace processing: if the same endpoint appears multiple times on the critical path, we sum them as long as their call chains are exactly the same. For example, in the previous $A_1B_1C_1B_3A_5D_1A_7$ critical path example, we merge the multiple occurrences of call paths A_* and $A_* \rightarrow B_*$. This merger discards the ordering relationship between events, which we do not need for further analysis.

5.2 Challenges with the Clock Drift

The span start and end times recorded in Jaeger traces are both callee’s local-machine time stamps converted to the standard UTC time. Machine clocks on two different physical machines drift [17, 49, 58] despite their periodic NTP-based synchronization. As a consequence of using local clocks, our critical path algorithm (if not corrected) can go wrong and sometimes lead to significant misattribution.

Span overlap problem: Figure 6 shows an ideal trace where the three spans A , B , and C are invoked one after another by the parent P . Most of the time should be attributed to the children. Figure 7 shows the trace for this example from our production, where the time recorded for the children spans have a small overlap; there is an overlap between the end of A and the start of B and the end of B and the start of C . In this case, the critical path

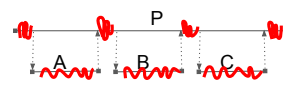


Figure 6: Ideal traces for a parent with three serialized children executions. Red lines show the corresponding critical path.

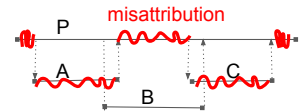


Figure 7: Actual traces due to clock drift. Red lines show the corresponding critical path.

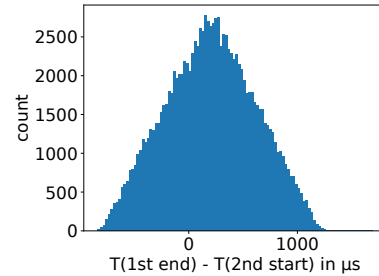


Figure 8: Distribution of time overlap recorded in Jaeger for two sequentially invoked RPCs. A positive value shows an overlap. The mean is $204.21\mu s$ and the max is $1696.00\mu s$.

is not attributed to span B and instead attributed to the parent. Due to the clock drift, more than 50% of our traces recorded this type of span overlaps causing misattribution in critical paths.

We conducted a detailed study on the impact of such clock drift. Figure 8 plots the time overlap recorded in Jaeger traces of two sequentially invoked RPCs sampled over 118K traces. A positive value shows overlap and a negative value shows non-overlap. More than 50% of samples show an overlap. The P50 overlap is $204\mu s$ and the maximum overlap is $1696\mu s$.

Based on this empirical observation, we tuned the happensBefore (A , B) part of our CP algorithm with the following relaxation:

- $A_{end} - threshold < B_{start}$, and
- No other children of the parent of P of A can start or end in the overlapped time range

The first condition allows a small `threshold` amount of overlap between the end of the previous span with the start of the next span. The second condition ensures that in the region of the allowed overlap, there is no other spawn and sync event, which ensures the parent-child serialization. The threshold is set to $1ms$.

Span overflow and underflow problems: In addition to the overlap, there can be overflow and underflow of child spans due to the clock drift. We enumerate these problems along with our pragmatic solutions below:

- A child span C may start before the start of the parent span P . In such cases, we truncate the start time of C till the start time of P . This may involve the recursive truncation of C ’s descendants.
- A child span C may end after the end of the parent span P . In such cases, we truncate the end time of C to the end time of P .

This may involve the recursive truncation of C 's descendant.

- Although rare, a child span C may end before the start time of parent span P . Similarly, a child span C may start after the end time of the parent span P . In these cases, we completely drop the subtree formed by C for CPA.

This tailoring fixed our CP algorithm. The total time truncation over millions of traces was under 5% giving us the confidence that a significant part of the data was retained.

5.3 Aggregating Critical Paths

While one trace can be compressed into its essential critical path and represented as a CCCT, it may not be representative. Hence, we need to inspect numerous traces to derive a “typical” shape of the critical path. Distinct traces may exhibit different critical paths based on many things, such as calling parameters, scheduling decisions, system load, time of the day, and network delays, to name a few. Hence, a *summary* of *typical* components on the critical path is desired.

To this end, we merge all critical paths (represented as CCCTs) into a weighted, *aggregate CCCT*. We follow the tree merging process done in HPCToolkit [13]. The aggregate CCCT succinctly summarizes all call paths leading to critical path nodes in all traces; it captures the quantitative aspect by associating higher weights to those call paths that are often on the critical path. The weights of the nodes in such a tree would be the summation of the weights of the constituent call paths. Specifically, we provide different percentiles (e.g., P50, P95, P99) of the latency values, which are widely used for QoS purposes. Figure 5 exemplifies this process.

5.4 Workflow for Continuous CPA

Figure 9 depicts the workflow followed by CRISP for performing critical path analysis of microservice traces for all endpoints. The components belonging to CRISP are marked by the outermost rectangular box.

All services are instrumented to produce Jaeger traces during their RPCs. The instrumentation is enabled across languages such as Go, Java, Node.JS, and Python. The RPCs emit Jaeger spans into a common data store, which can be queried via SQL-style queries.

The CRISP workflow runs as a daily job. The workflow begins by collecting a list of endpoints. Each endpoint can be handled in parallel. Hence, we dedicate a handful of machines that shard the list of endpoints among them.

For each endpoint, CRISP queries the Jaeger data store (via `sawmill-query`) service to fetch a list of traceIDs. This query is set up to obtain the last two weeks' worth of traces. We then use these traceIDs to fetch the actual JSON traces (`jaeger-query`) service. We exploit IO parallelism here to fetch many traces concurrently.

We compute the critical path over each trace in parallel using the map-reduce paradigm. The set of critical paths obtained is

fed into an aggregating process that summarizes and produces the daily critical path report for each endpoint (top-down analysis) and also produces overall metrics aggregated over all endpoints (bottom-up analysis). The results are injected into blob storage that can be easily navigated by a varied set of users, including service owners, performance engineers, and capacity managers. An offline anomaly detection model is also trained per endpoint result.

6 CRISP Features

We have developed tools to inspect critical paths for top-down performance analysis of specific endpoints, bottom-up analysis over all endpoints, and automatic anomaly detection over traces. We describe these features in this section.

6.1 Top-Down Analysis

We store the results of our CPA for each endpoint into profiles for investigation by service owners. CRISP provides the following means of visualization of CPA over each endpoint.

Flame graph: Flame graph [31] is a powerful way to visualize hierarchical call paths arising from profiling. The interactive visualization is easier to digest and investigate. Since we maintain the summarized critical paths as aggregate CCCTs, which are formed of many weighted call paths, it naturally avails itself to be represented as a flame graph.

If we chose all traces to represent a single flame graph, the critical path found in P99 latencies may dominate the flame graph and mask the other common cases. For that reason, we show three different flame graphs for different percentiles of latency values (e.g., P50, P95, and P99). We also produce differential flame graphs [30] that show how the critical paths change between two percentile values.

Heat map: Flame graphs are useful for navigating call chains but developers sometimes need access to an actual Jaeger trace that represents a given data so that they can inspect it in further detail. For this reason, CRISP provides the heat map view (see Figure 10), where the rows are the endpoints and the columns represent individual traces. Each cell in the heat map represents the exclusive time on the critical path and each cell is gradient colored based on its contribution (exclusive time) to the total latency. In this view, we collapse the call paths and accumulate the metrics from all call paths, reaching the same endpoint in a single row. However, for exploration, the developers have access to the top 5 call chains (not shown) for each endpoint, which is available by hovering over any row. In this view, the user can also choose percentile values and inclusive or exclusive metrics to sort the rows. Each column is also sorted by a high to low contribution for a given chosen metric. Selecting any trace takes the user to the Jaeger-UI to inspect the trace.

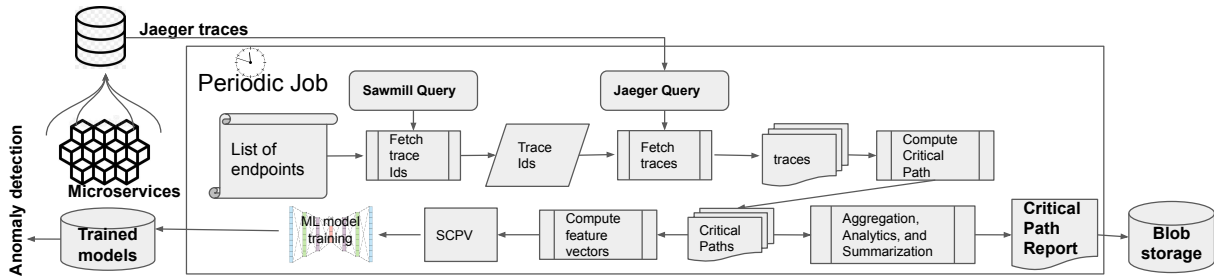


Figure 9: Schematic diagram of CPA over Jaeger traces.

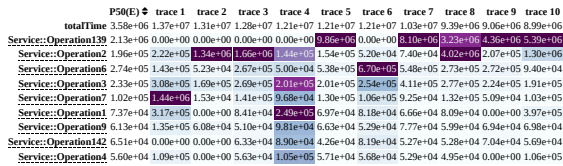


Figure 10: Example heat map from 1000 traces. The result is sorted by the P50 percentile value of the exclusive time of each operation. Each cell is the accumulated time in μ s.

6.2 Bottom-Up Analysis

The objective of the bottom-up analysis is to derive insights from *all* endpoints and to bubble up those interior APIs improving which will improve many endpoints. The bottom-up analysis is a data-intensive process and needs access to critical paths from all endpoints. For this reason, we retain the aggregate CCCT computed for each endpoint from the top-down process, along with some additional statistics related to the overall graph structure. Once all endpoints are processed, the bottom-up analysis runs; it aggregates the statistics from each endpoint and quantifies the impact of each API over all other endpoints. The output of the bottom-up analysis is a descending priority list of top APIs that are often in many endpoints. Additionally, the bottom-up analysis produces various histograms over all traces taken together, which include the total number of times any API appears in any graph, the total number of times an API appears on the critical path, the number of unique APIs on the critical path, the critical path length, and the maximum degree of concurrency in a trace, among others. These graphs are intended to inform infrastructure and hardware engineers to better understand the current needs of our systems and aid capacity planning for the future.

6.3 Anomaly Detection

We also employ CRISP to pinpoint whether a new incoming trace (for a given endpoint) deviates from the normal execution behavior. For this purpose, we have trained a machine learning model and used it for inference.

During the offline training, we encode the critical path (CCCT) for each trace of an endpoint into feature vectors,



Figure 11: An example CCCT (left), the letters indicate name and the numbers indicate the exclusive time on the span. The corresponding SCPV (right).

which we call service critical path vectors (SCPV). We feed several SCPVs into an autoencoder to learn the normal execution pattern of the given service. During the online inference, the learned model will infer whether the given new trace is abnormal or not based on an anomaly score.

The architecture design, training, and inference of the autoencoder are derived from TraceAnomaly [39], which is the state-of-the-art framework for anomaly detection in microservices trace. The neural architectural details are described in Appendix B. The key difference between CRISP and TraceAnomaly is in the data encoding. TraceAnomaly uses a service trace vector (STV) which encodes every path in the trace and, in contrast, CRISP encodes only on the call paths for those spans that are on the critical path spans.

SCPV encoding: Figure 11 exemplifies encoding the critical path present as a CCCT into an SCPV. For each node in CCCT, it assigns weights based on its exclusive execution time. Notice that endpoint *C* occurs twice on the critical path, thus it is also encoded twice in the SCPV, given the call chain is different. The training set is a 2D matrix where each column is a feature (call path) and each row is the feature values of a given trace.

Using the call paths of spans only on the critical path, compared with the prior work that used all call paths in the entire graph, offers significant benefits. It reduces the feature dimensions; it reduces the training and inference time; and, most importantly, it improves the model accuracy. The impact of the CCCT-based encoding is substantial and evaluated in Section 7.3.

7 Experience and Evaluation

In Section 7.1, we describe one of our findings by applying the top-down analysis of CRISP at Uber, in Section 7.2 we

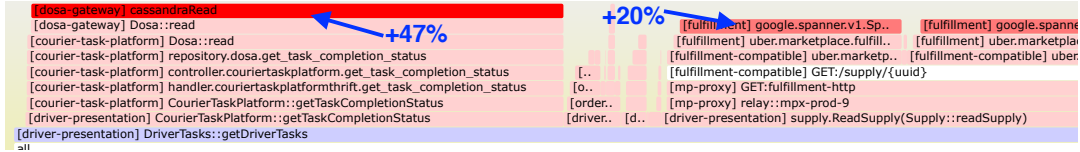


Figure 12: Differential flame graph for the `getDriverTask` endpoint. Red colors indicate the growth from P50 critical paths to P95 critical paths.

show valuable characteristics of microservices at Uber by applying the bottom-up analysis of CRISP. In Section 7.3, we empirically evaluate the anomaly detection capability of CRISP and in Section 7.4 we describe how we employed CRISP in guiding future hardware selection to reduce tail latency in our services.

7.1 Tail Latency Investigation via Top-Down Analysis

`getDriverTasks` is a business-critical endpoint in the `driver-presentation` service responsible for returning the task plan that a driver needs to perform. A sample task plan could be: passenger mask check, pickup passenger, pickup food, drop off passenger, and drop off food. This endpoint assembles the task plan and enriches it by calling numerous other microservices such as `courier-task-platform`.

Figure 12 shows the *differential* flame graph for the `getDriverTask` endpoint. The graph plots a difference between the critical paths seen in the traces with the P50 latency vs. P95 latency for the `getDriverTask` endpoint. The red-colored boxes show the growth in percentage time spent in P95 with regards to P50. The `getTaskCompletionStatus` API was absent in the P50 traces, whereas it occupies 47% of the total execution in P95 traces, contributing to the same amount of addition to the tail latency. This endpoint dependency makes a call to Cassandra—an expensive database read. Based on this insight from CRISP’s differential flame graph views, we identified the root cause of performance variance and high tail latency. We recommend caching with timestamp filtering optimization as opposed to a database read to reduce the tail latency.

Trace processing overheads: Table 1 shows the overhead of analyzing the `getDriverTasks` endpoint discussed in this section running on 16 cores of an Intel Xeon Skylake machine clocked at 2.4 GHz.

Table 1: Overhead of top-down analysis of `getDriverTasks`.

Num Traces	Trace size	Processing time	Memory usage
10k	6.8 GB	48 sec	2.1 GB
20k	14 GB	109 sec	4.2 GB
40K	28 GB	232 sec	8.5 GB
80K	56 GB	553 sec	17.6 GB

Sparse sampling vs. quality of CPA: We observed that the sampling rate does not qualitatively affect the aggregate critical path results. We conducted an experiment where we first produced an aggregate critical path from 1 million traces.

We also produced critical paths from randomly sampled 100K and 10K traces from the same data set. We noticed that the attribution of the top 20 services on the critical path, whether for 10K or 100K samples, was essentially the same as the one produced from 1M traces.

7.2 Systemic Insights via Bottom-Up Analysis

In this section, we show the result of running CRISP with bottom-up analysis on the collected trace dataset and some insight associated with the data. The dataset includes more than 1 million traces, ~4k services, and ~40k endpoints. It takes around 4 hours on 32-cores of a Intel Xeon Skylake machine clocked at 2.4 GHz.

Total RPCs per request: Figure 13 is a histogram of the total number of RPCs made per request, which is same as the total number of spans in a trace. On average there are 112 spans in a trace. However, there exist several large ones with a maximum of 275K spans. Such scale brings significant challenges for the developer to debug without proper reduction of the graph size.

Total endpoints in a trace: Figure 14 is a histogram of the total number of unique endpoints found in each trace. At most each trace has 1400 unique endpoints.

Latency distribution: Figure 15 plots the histogram of latencies observed in each of ~1M traces. The tail is several orders of magnitudes longer than the mean or median.

RPC depth: Figure 16 is a histogram of the longest call chain found in each trace. The depth of the call chain is another measure of the complexity of traces. The average RPC depth is 8.5. The maximum observed depth is 36.

Unique caller: Figure 17 is a histogram of the number of the unique callers for each endpoint across one million traces. The number differs wildly as the mean value is just above 2 but the maximum value is 620.

Degree of concurrency: Figure 18 is a histogram of the maximum number of spans that overlap in time in each trace. This number gives the degree of concurrency (and hence a measure of the complexity) in our traces. Overall, the microservices show a high degree of concurrency. On average, the degree of concurrency is 21. The degree of concurrency often grows to 100s for more complicated services. The maximum degree of concurrency we observed in ~1M traces was 3076.

Total RPCs on the critical path: Figure 19 is a histogram of the number of spans on the critical paths, which counts the number of RPCs made on the critical path. Besides a few outliers, the length of the critical path is short. On average, there are 33

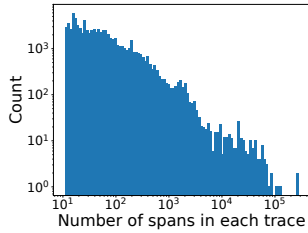


Figure 13: Histogram of the number of spans per trace.

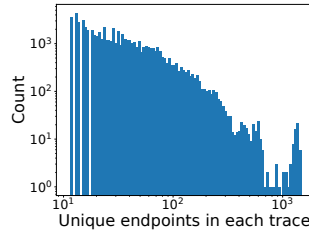


Figure 14: Histogram of number of unique endpoints per trace.

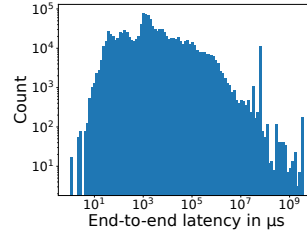


Figure 15: Distribution of latency among all traces.

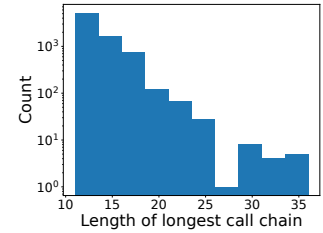


Figure 16: Histogram of longest call chain per trace.

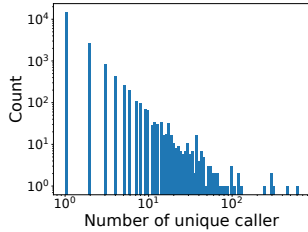


Figure 17: Histogram of the number of unique caller for each endpoint.

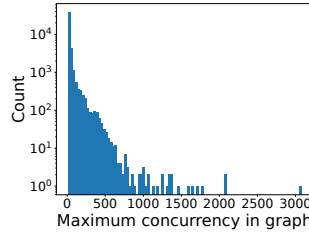


Figure 18: Histogram of the degree of the concurrency (max no. of overlapping spans) per trace.

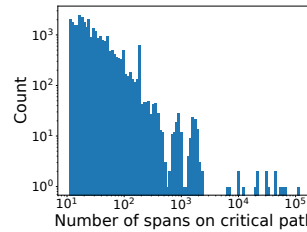


Figure 19: Histogram of the number of spans on the critical path per trace.

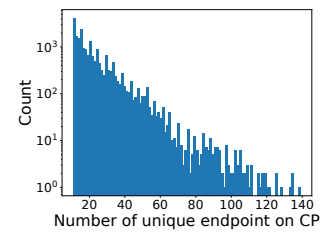


Figure 20: Histogram of the number unique endpoints on the critical path per trace.

RPCs on the critical path (in contrast, the entire graph in Figure 13 shows 112-275K RPCs in traces). The short critical path length allows the developer to investigate and debug easily.

Endpoints on the critical path: Figure 20 is a histogram of the unique endpoints on each critical path. Compared with the number of endpoints in the entire trace (Figure 14), the number of the endpoints on the critical path is an order of magnitude smaller (the maximums are 1400 vs. 140). The 10x size-reduction matches our observation of the 6 services we test for anomaly detection.

7.3 Empirical Analysis of Anomaly Detection

Here, we will evaluate CRISP’s anomaly detection on six critical endpoints.

Methodology: We collect traces for six microservices in real production over a 14-day period. The training data for each case includes 20,000 traces and the testing data has 500 unseen traces for normal and abnormal data. To generate abnormal inference data, we drop 20% of the nodes in the graph and randomly shuffle the duration of the nodes as described in [26,39,48]. We did not use real anomalous traces for evaluation since we do not have a large number of labelled anomalous traces (i.e., we have a lot of false negatives). Also, the labeled data contains false positives and coordinating with hundreds of developers to verify the veracity of labeling is non-trivial.

We use TraceAnomaly [39] as the baseline against which we compare our results. We adopt the same architecture of the autoencoder and reuse their code. The main difference is that we use CRISP to preprocess the trace before feeding it into the autoencoder so that only paths appearing on the critical path information are included. A fundamental assumption is that any

noticeable difference in the trace must impact the critical path.

Hardware: We use two machines in our evaluation: a CPU-only machine with 256 GB memory and a CPU+GPU machine with 128 GB memory. Most of the experiment is done on a machine with GPUs. It has 2 Quadro RTX 5000 GPUs and 2 socket Intel Xeon Gold 5218 CPU at 2.30GHz. The CPU machine has 2 sockets with Intel Xeon Silver 4214 CPU at 2.20GHz. Both machines run on Linux 4.14. The reason to use two machines is that for some experiments, the training data for TraceAnomaly cannot fit the GPU memory, whereas CRISP’s training data always fits on GPU memory. In such cases, for a fair comparison, we also run the experiment on the 256 GB CPU-only.

Table 2 shows the empirical evaluation results of anomaly detection on 6 large online services at Uber. It captures the essential features such as the number of RPCs, unique endpoints, and call path diversity in these services. It also shows the training and inference time with both STV (prior art from TraceAnomaly) and SCPV (our work) data. Finally, the last 4 columns present the model accuracy in terms of precision and recall. In summary, using critical path via CRISP reduces the training time and inference time and improves the recall performance on top of the state of the art.

Training speedup: From the table, we can observe that CRISP offers up to 22× speedup for training compared with TraceAnomaly. Even the smallest speedup is more than 50%.

The reason for the speedup is that the training data from CRISP (SCPV) is one magnitude smaller than TraceAnomaly (STV) up to 25× for Service 6. The number of unique call paths on the critical path is significantly smaller than the total number of call paths in the entire graph (also see Figures 13-20). Furthermore, when the number of the trace and the dimension

Table 2: Evaluation results for large online services. Inference time is measured with 1000 traces. (TA*=TraceAnomaly.)

	No. of Unique endpoints	Max no. of spans	No. of callpaths/features		Training Time		CRISP training speedup	Inference Time		CRISP inference speedup	Precision		Recall	
			STV	SCPV	TA*	CRISP		TA*	CRISP		TA*	CRISP	TA*	CRISP
Service 1	214	1429	5117	1186	70m (GPU)	46m (GPU)	1.52X	2.24s (GPU)	1.21s (GPU)	1.85X	1.0	0.998	0.986	0.992
Service 2	969	1724	9725	1860	100m (GPU)	50m (GPU)	2.00X	3.54s (GPU)	1.40s (GPU)	2.54X	1.0	1.0	0.958	0.984
Service 3	734	5320	20321	2154	150m (GPU)	50m (GPU)	3.00X	5.64s (GPU)	1.36s (GPU)	4.15X	1.0	1.0	0.5	0.982
Service 4	912	20001	25347	2715	1184m (CPU)	56m (GPU) 219m (CPU)	21.14X (GPU) 5.41X (CPU)	56.67s (CPU)	1.56s (GPU) 9.26s (CPU)	36.33X (GPU) 6.12X (CPU)	1.0	1.0	0.928	0.978
Service 5	768	6562	26404	2336	811m (CPU)	51m (GPU) 177m (CPU)	15.90X (GPU) 4.58X (CPU)	42.90s (CPU)	1.36s (GPU) 5.81s (CPU)	31.54X (GPU) 7.38X (CPU)	1.0	0.998	0.5	0.982
Service 6	1477	10992	28968	1151	1305m (CPU)	46m (GPU) 148m (CPU)	27.77X (GPU) 8.82X (CPU)	78.88s (CPU)	1.18s (GPU) 4.48s (CPU)	66.85X (GPU) 17.61X (CPU)	1.0	1.0	0.912	0.977

of the feature vector is large, the size of the training data of TraceAnomaly can easily exceed the memory of the GPU, which makes it unable to train. For such cases (Service 4, 5, and 6), we can still see more than $4\times$ speedup even if we train both TraceAnomaly and CRISP on CPU machines. When CRISP is trained on the GPU machine, the speedup can easily exceed $15\times$. The faster training allows for more practical deployment.

Inference speedup: Similar to training speedup, the reduction in inference data size leads to a faster inference of CRISP. The smallest speedup is more than $1.85\times$ whereas the largest speedup is over $66\times$. This lower latency allows us to batch many inferences together to exploit GPU throughput.

Precision: From Table 2, we can see that both TraceAnomaly and CRISP are capable of detecting the abnormal trace accurately. Autoencoders are capable of capturing the complex pattern of the graph. TraceAnomaly works slightly better than CRISP on 2 services, but overall accuracy is very high for both methods.

Recall: The recall is the part that differentiates the quality of results between TraceAnomaly and CRISP. Recall measures how many of the actual positives the model captures through labeling it as positive, (i.e., $\frac{True_Positive}{True_Positive+False_Positive}$). When the recall is closer to 1, it indicates that the model makes fewer false-positive predictions (an anomaly in this case). From Table 2, it is clear that CRISP outperforms TraceAnomaly by a noticeable margin. Particularly for Service 3 and 5, half of the positive prediction of the anomaly is false, meaning all normal traces for inference are labeled abnormal by TraceAnomaly. To make sure the prediction is actually incorrect, we asked the service owners and verified that the normal inference testing traces are not showing any abnormal behaviors. On the contrary, CRISP’s recall is close to 1. For Service 1 and 2, the performance of CRISP is slightly better than TraceAnomaly, as both models make relatively accurate predictions. CRISP shows more than 5% improvement for Service 4 and 6.

CRISP produces superior results on services with a large number of call paths. For instance, there are 912 endpoints in Service 4 but the total call paths is 25,347. Since there is more diversity among the shapes of the call chains on the entire graph, the SCPV encoding fails to capture its full variety; consequently, unseen call paths easily trigger a false positive in TraceAnomaly. In contrast, the critical path remains

fairly stable when trained over a large corpus of traces, and consequently CRISP has fewer false positives.

7.4 CPA in Hardware Selection

In addition to the parent-child transitive relationships and times, Jaeger traces also contain additional information, such as the hostname on which the span was executed. Uber’s data center consists of diverse hardware CPU SKUs. Services can be installed on different hardware versions. Hence, an API may run on different hardware on different requests.

We collected the critical path for one of our important services using CRISP and identified that a downstream operation was on the critical path. We further clustered the samples from the profiles by the CPU versions on which they were running. The violin plot in Figure 21 in Appendix A shows how the latencies vary on 2 prominent CPU SKUs: Intel Xeon Silver 4212 running at 2.2 GHz (SKU-A) and Intel Xeon Silver 4212R running at 2.4 GHz (SKU-B). The two SKUs are identical (same vendor, microarchitecture, cache size, etc.) with the only exception being that their CPU clock speeds are different. This mild (9%) difference in the clock speed has a profound impact on the behavior of the plotted service. The P50 value for SKU-A is 15% higher than that on SKU-B. Moreover, the tail latency on SKU-A is 1.5x higher than the one on SKU-B.

To summarize, a slightly faster CPU clock proves to have a significant impact on reducing the tail latency and overall latency. This difference has a significant impact on the overall capacity allocation since tail latency (e.g., P95) is often used in capacity allocation. This observation demands further, systematic investigation into classifying critical path components as CPU SKU sensitive vs. insensitive; also, such categorization helps data center-wide microservice schedulers to favor SKU-sensitive services on the critical path onto the SKUs where they exhibit superior performance.

8 Related Work

Critical Path Analysis (CPA) has been extensively explored in the shared-memory parallel programming paradigm [13, 20, 25, 40, 50, 54, 55, 59, 62] but less explored in distributed parallel systems. Unlike shared-memory and struc-

tered parallel programs, microservices use distributed parallel computing environments and are unstructured in nature.

Barford and Crovella [16] utilize critical path analysis for profiling and understanding TCP transactions and improve data transfer latency in web applications; however their scale is significantly smaller than the 4K services deployed over millions of CPU cores that we handle. Bohem et al. [18] employ tracing and CPA for MPI programs in HPC environments; this approach has not been employed in microservice environments. Kaldor et al. [33] develop an end-to-end tracing system (Canopy) for tracking requests from web-browsers/mobile to backend services; it handles billions of traces. A distinguishing feature of CRISP compared with Canopy is the use of CPA, which significantly reduces the data needed for analysis.

Qiu et al. [48] propose a fine-grained resource management framework based on microservice traces using CPA. They employ the insights for scheduling and other resource management to reduce CPU utilization. However, their work does not cover industry-scale deployment; they also do not facilitate performance bug or anomaly detection and cannot provide bottom-up system-wide performance insights.

Fields et al. [24] explore a hardware predictor to analyze the criticality of instructions by using CPA and use it to guide dynamic instruction scheduling. Venkataramani et al. [55] propose Global Critical Path (GCP) to predict system-level performance and optimize the performance of highly concurrent self-timed circuits. These approaches rely on the precise last arriver information, which is readily available in these cases. Our critical path computation in microservices also depends on knowing the last arriver. Unlike the aforementioned approaches, we do not have direct access to the last arriver in our distributed system. As a result, we need to use clock information from different hosts and adjust for clock skew to heuristically infer the last arriver.

Multiple tools have been developed to profile and debug large distributed and parallel systems. `lprof` [64] constructs request flow from logs and it is as good as the quality of logs; it has not been evaluated on microservices; it also does not provide CPA and hence suffers from a voluminous noisy data. Mace et al. [41] developed Pivot as a dynamic, extensible tracing system for inter-operating applications. Pivot employs a happen-before relationship between events to establish causality. Pivot does not build a critical path and hence pays equal attention to any causal relationship unlike CRISP. Chow et al. [19] build a system that utilizes a large number of request traces to validate hypotheses about causal relationships. Edgar [2] provides a summarized view of request traces, logs, and metadata in distributed systems. It does not employ sophisticated analyses or automated anomaly detection.

Several works have focused on microarchitectural aspects of microservices [34, 42, 52, 60, 61]. Most of these works are focused on how microservices utilize microarchitectural features, but ignore the end-to-end user request; in contrast, CRISP takes a higher-level approach and looks at the entire

flow of requests through a chain of services.

Multiple works have studied anomaly detection in distributed systems. Liu et al. [39] use Deep Bayesian Network to detect the performance anomaly in an unsupervised manner. They utilize machine learning to learn the normal behavior pattern of the given dependency graph and try to detect the anomaly online. Gan et al. [26] propose a root cause analysis system for large-scale microservices using machine learning. The system uses Conditional Variational Autoencoders (CVAE) [51] to automatically generate the counterfactual training data. These approaches have used the entire call graph, leading to significant training and inference time. In contrast, CRISP uses only the critical path(s), leading to dramatic speedups while producing higher quality results.

9 Conclusions and Future Work

Microservices are the preferred architecture choice in modern service-oriented software systems. Large-scale microservices have tens of thousands of endpoints with complex, nested, and asynchronous. Prior work in profiling microservices has either focused on tracing techniques, which produce a lot of data, but lack in delivering insights, or on micro-architectural optimization within a service, ignoring the full picture of the life of a request through myriad services. This paper develops a tool, CRISP, which uses critical path analysis (CPA) over RPC traces to bubble-up interesting activities and discard noisy events. CRISP provides rich developer insights both for service owners and infrastructure engineers. In a short three-month deployment period, CRISP's analyses have sifted over 4,000+ services, 40,000+ endpoints, hundred of millions of traces, and tens of terabytes of data at Uber; as a result, CRISP has bubbled-up profiling results that helped developers understand and optimize important services. Employing the critical path, as opposed to the whole RPC trace, speeds up the training of models and on-the-fly inference for anomaly detection while also producing noticeably higher quality results.

Our future work involves enhancing CRISP to address other use cases such as setting the TTL values for downstream calls and bubbling up those downstream services that often return errors. We plan to expand our anomaly detection to include developers in the loop and improve traces with labelled data.

Availability

Parts of the code of this work are open-sourced [1].

Acknowledgement

This material is based upon work supported by the National Science Foundation under Grants No. 2006542 and 1763699. We thank our shepherd and the anonymous reviewers for their feedback. We also thank William Fulton and Sara Wilmes-Reitz for their help with proof-reading a draft version of this paper.

References

- [1] CRISP: Critical Path Analysis of Microservice Traces. <https://github.com/uber-research/CRISP>.
- [2] Edgar: Solving Mysteries Faster with Observability. <https://netflixtechblog.com/edgar-solving-mysteries-faster-with-observability-ela76302c71f>. (Accessed on 11/30/2021).
- [3] Evolving Schemaless into a Distributed SQL Database. <https://eng.uber.com/schemaless-sql-database/>. (Accessed on 01/12/2022).
- [4] Fast and Reliable Schema-Agnostic Log Analytics Platform. <https://eng.uber.com/logging/>. (Accessed on 01/12/2022).
- [5] Heat Map. https://en.wikipedia.org/wiki/Heat_map. (Accessed on 01/11/2022).
- [6] Jaeger: open source, end-to-end distributed tracing. <https://www.jaegertracing.io/>. (Accessed on 12/01/2021).
- [7] perf (linux) - wikipedia. [https://en.wikipedia.org/wiki/Perf_\(Linux\)](https://en.wikipedia.org/wiki/Perf_(Linux)). (Accessed on 01/12/2022).
- [8] Uber's Fulfillment Platform: Ground-up Re-architecture to Accelerate Uber's Go/Get Strategy. <https://eng.uber.com/fulfillment-platform-rearchitecture/>. (Accessed on 01/12/2022).
- [9] User Datagram Protocol. https://en.wikipedia.org/wiki/User_Datagram_Protocol. (Accessed on 01/11/2022).
- [10] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page, 2013.
- [11] Profiling Go Programs. <https://blog.golang.org/pprof>, 2013.
- [12] Head-based and tail-based sampling, rate-limiting. <https://opentelemetry.uptrace.dev/guide/sampling.html#introduction>, April 2022.
- [13] ADHANTO, L., BANERJEE, S., FAGAN, M., KRENTEL, M., MARIN, G., MELLOR-CRUMMEY, J., AND TALLENT, N. R. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.
- [14] AMMONS, G., BALL, T., AND LARUS, J. R. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1997), PLDI '97, Association for Computing Machinery, p. 85–96.
- [15] APACHE FLINK TEAM. Apache Flink: Stateful Computations over Data Streams. <https://flink.apache.org/>.
- [16] BARFORD, P., AND CROVELLA, M. Critical path analysis of TCP transactions. *ACM SIGCOMM Computer Communication Review* 30, 4 (2000), 127–138.
- [17] BLUEMATADOR. Time Drift (NTP). <https://www.bluematador.com/docs/troubleshooting/time-drift-ntp>.
- [18] BÖHME, D., GEIMER, M., ARNOLD, L., VOIGTLÄENDER, F., AND WOLF, F. Identifying the Root Causes of Wait States in Large-Scale Parallel Applications. *ACM Trans. Parallel Comput.* 3, 2 (jul 2016).
- [19] CHOW, M., MEISNER, D., FLINN, J., PEEK, D., AND WENISCH, T. F. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 217–231.
- [20] CURTSINGER, C., AND BERGER, E. D. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 184–197.
- [21] DELIMITROU, C., AND KOZYRAKIS, C. Amdahl's Law for Tail Latency. *Commun. ACM* 61, 8 (jul 2018), 65–72.
- [22] EPANECHNIKOV, V. A. Non-parametric estimation of a multivariate probability density. *Theory of Probability & Its Applications* 14, 1 (1969), 153–158.
- [23] FERDMAN, M., ADILEH, A., KOCHBERBER, O., VOLOS, S., ALISAFABEE, M., JEVDJIC, D., KAYNAK, C., POPESCU, A. D., AILAMAKI, A., AND FALSAFI, B. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, Association for Computing Machinery, p. 37–48.
- [24] FIELDS, B., RUBIN, S., AND BODIK, R. Focusing processor policies via critical-path prediction. In *Proceedings 28th Annual International Symposium on Computer Architecture* (2001), IEEE, pp. 74–85.
- [25] FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1998), PLDI '98, Association for Computing Machinery, p. 212–223.

- [26] GAN, Y., LIANG, M., DEV, S., LO, D., AND DELIMITROU, C. Sage: practical and scalable ML-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), pp. 135–151.
- [27] GLUCK, A. Introducing Domain-Oriented Microservice Architecture. <https://eng.uber.com/microservice-architecture/>.
- [28] GOLDBERG, Y. Scaling Gilt: from Monolithic Ruby Application to Distributed Scala Micro-Services Architecture. <https://www.infoq.com/presentations/scale-gilt>.
- [29] GRAHAM, S. L., KESSLER, P. B., AND MCKUSICK, M. K. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (New York, NY, USA, 1982), SIGPLAN '82, Association for Computing Machinery, p. 120–126.
- [30] GREGG, B. Differential Flame Graphs. <https://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html>.
- [31] GREGG, B. The flame graph. *Communications of the ACM* 59, 6 (2016), 48–57.
- [32] HAQUE, M. E., HE, Y., ELNIKETY, S., NGUYEN, T. D., BIANCHINI, R., AND MCKINLEY, K. S. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2017), MICRO-50 '17, Association for Computing Machinery, p. 625–638.
- [33] KALDOR, J., MACE, J., BEJDA, M., GAO, E., KUROPATWA, W., O'NEILL, J., ONG, K. W., SCHALLER, B., SHAN, P., VISCOMI, B., ET AL. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 34–50.
- [34] KANEV, S., DARAGO, J. P., HAZELWOOD, K., RANGANATHAN, P., MOSELEY, T., WEI, G.-Y., AND BROOKS, D. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015), pp. 158–169.
- [35] KINGMA, D. P., AND WELLING, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).
- [36] KRAMER, S. The Biggest Thing Amazon Got Right: The Platform. <https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/>, October 2011.
- [37] KREPS, J., NARKHEDE, N., RAO, J., ET AL. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB* (2011), vol. 11, pp. 1–7.
- [38] LI, J., SHARMA, N. K., PORTS, D. R. K., AND GRIBBLE, S. D. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, Association for Computing Machinery, p. 1–14.
- [39] LIU, P., XU, H., OUYANG, Q., JIAO, R., CHEN, Z., ZHANG, S., YANG, J., MO, L., ZENG, J., XUE, W., ET AL. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)* (2020), IEEE, pp. 48–58.
- [40] LIU, X., MELLOR-CRUMMEY, J., AND FAGAN, M. A New Approach for Performance Analysis of OpenMP Programs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (New York, NY, USA, 2013), ICS '13, Association for Computing Machinery, p. 69–80.
- [41] MACE, J., ROELKE, R., AND FONSECA, R. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 378–393.
- [42] MARS, J., TANG, L., HUNDT, R., SKADRON, K., AND SOFFA, M. L. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-Locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2011), MICRO-44, Association for Computing Machinery, p. 248–259.
- [43] MAURO, T. Adopting Microservices at Netflix: Lessons for Architectural Design. <https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/>, Feb 2015.
- [44] MIRHOSSEINI, A., SRIRAMAN, A., AND WENISCH, T. F. Enhancing Server Efficiency in the Face of Killer Microseconds. In *Proceedings of the 25th International Symposium on High-Performance Computer Architecture (HPCA '19)* (2019), IEEE, pp. 185–198.
- [45] NADAREISHVILI, I., MITRA, R., McLARTY, M., AND AMUNDSEN, M. *Microservice Architecture: Aligning Principles, Practices, and Culture*, 1st ed. O'Reilly Media, Inc., 2016.
- [46] OPENTRACING DEVELOPERS. OpenTracing: Span. <https://opentracing.io/docs/overview/spans/>.

- [47] OPENTRACING DEVELOPERS. OpenTracing: What is Distributed Tracing? <https://opentracing.io/docs/overview/what-is-tracing/>.
- [48] QIU, H., BANERJEE, S. S., JHA, S., KALBARCZYK, Z. T., AND IYER, R. K. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 805–825.
- [49] S V, S. Time drift monitoring: Troubles of unsynchronized servers - site24x7 blog. <https://www.site24x7.com/blog/time-drift-monitoring-troubles-of-unsynchronized-servers>. (Accessed on 06/02/2022).
- [50] SCHARDL, T. B., KUSZMAUL, B. C., LEE, I.-T. A., LEISERSON, W. M., AND LEISERSON, C. E. The Cilkprof Scalability Profiler. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2015), SPAA '15, Association for Computing Machinery, p. 89–100.
- [51] SOHN, K., LEE, H., AND YAN, X. Learning structured output representation using deep conditional generative models. *Advances in neural information processing systems 28* (2015), 3483–3491.
- [52] SRIRAMAN, A., DHANOTIA, A., AND WENISCH, T. F. SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale. In *Proceedings of the 46th International Symposium on Computer Architecture* (2019), Association for Computing Machinery, p. 513–526.
- [53] SRIRAMAN, A., LIU, S., GUNBAY, S., SU, S., AND WENISCH, T. F. Deconstructing the Tail at Scale Effect Across Network Protocols. *The Annual Workshop on Duplicating, Deconstructing, and Debunking* (2016).
- [54] TALLENT, N. R., AND MELLOR-CRUMMEY, J. M. Effective performance measurement and analysis of multithreaded applications. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (2009), pp. 229–240.
- [55] VENKATARAMANI, G., BUDI, M., CHELCEA, T., AND GOLDSTEIN, S. C. Global critical path: A tool for system-level timing analysis. In *Proceedings of the 44th annual Design Automation Conference* (2007), pp. 783–786.
- [56] VILLAMIZAR, M., GARCÉS, O., CASTRO, H., VERANO, M., SALAMANCA, L., CASALLAS, R., AND GIL, S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)* (2015), pp. 583–590.
- [57] WESTFALL, P. H., AND YOUNG, S. S. *Resampling-based multiple testing: Examples and methods for p-value adjustment*, vol. 279. John Wiley & Sons, 1993.
- [58] WIKIPEDIA. Clock drift. https://en.wikipedia.org/wiki/Clock_drift.
- [59] YANG, C.-Q., AND MILLER, B. Critical path analysis for the execution of parallel and distributed programs. In *[1988] Proceedings. The 8th International Conference on Distributed* (1988), pp. 366–373.
- [60] YANG, H., BRESLOW, A., MARS, J., AND TANG, L. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2013), ISCA '13, Association for Computing Machinery, p. 607–618.
- [61] YASIN, A. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2014), IEEE, pp. 35–44.
- [62] YOGA, A., AND NAGARAKATTE, S. Parallelism-centric what-if and differential analyses. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019), pp. 485–501.
- [63] ZHANG, Y., MEISNER, D., MARS, J., AND TANG, L. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (2016), pp. 456–468.
- [64] ZHAO, X., ZHANG, Y., LION, D., ULLAH, M. F., LUO, Y., YUAN, D., AND STUMM, M. lprof: A non-intrusive request flow profiler for distributed systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 629–644.

A Violin Plot for Hardware Selection

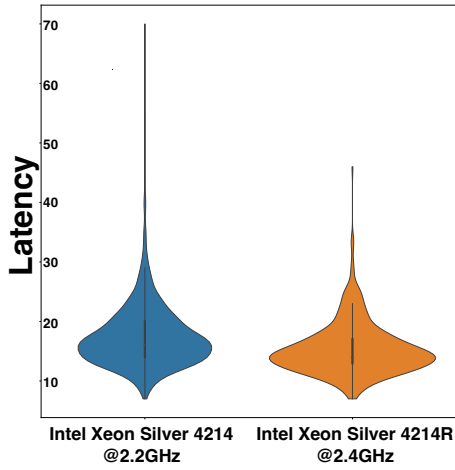


Figure 21: Violin plots of the exclusive execution time of a critical path operation with two different CPUs. The latency is in μs .

B Autoencoder Model Architecture

We choose the Deep Bayesian Network for anomaly detection given it is capable of learning complex patterns from the trace. We adopt the model from TraceAnomaly [39], which is the state-of-the-art framework for microservice trace based anomaly detection. Specifically, we adopt Variational Auto-Encoder (VAE) [35] to model the distribution pattern from the normal execution. VAE is an unsupervised learning that does not require a label, which can be expensive to obtain in our setting due to the volume of traces. Figure 22 depicts the architecture of VAE. It has three components: encoder, posterior flow, and decoder.

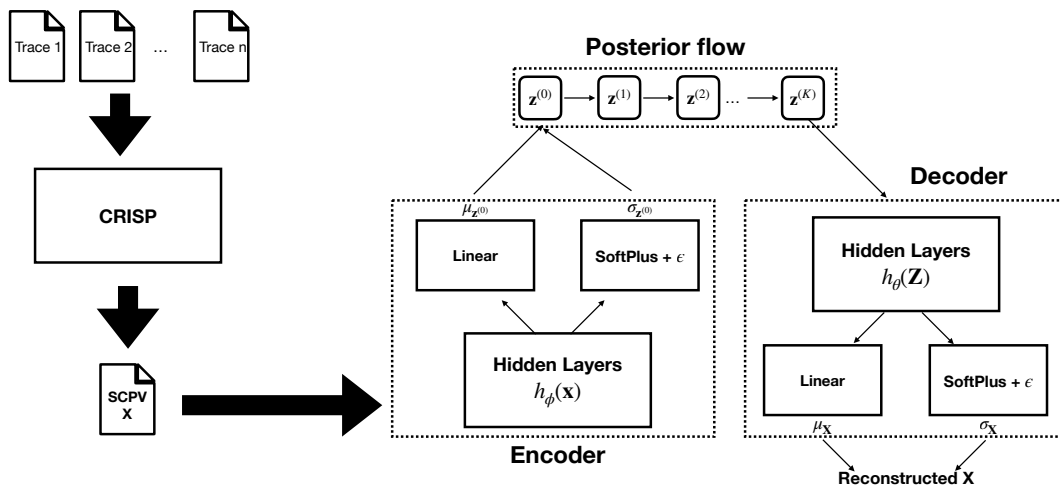


Figure 22: Architecture of neural network for anomaly detection.

The encoder contains 1 hidden layer ($h_\phi(\mathbf{x})$) to learn the hidden features of SCPV. The goal is to learn the mean $\mu_{\mathbf{z}^{(0)}}$ and the standard deviation $\sigma_{\mathbf{z}^{(0)}}$ of the SCPV. $\mathbf{z}^{(0)}$ is sampled from diagonal Gaussian $\mathcal{N}(\mu_{\mathbf{z}^{(0)}}, \sigma_{\mathbf{z}^{(0)}} \mathbf{I})$ and served as the latent variable to fit the distribution. ϵ is a small constant vector that has been introduced to avoid numerical issues during the training [39]. SoftPlus is defined as $\text{SoftPlus}(\mathbf{x}) = \log(1 + \exp(\mathbf{x}))$.

For the next step, posterior flow allows the network to learn more complex patterns of the trace. The input is $\mathbf{z}^{(0)}$ and after passing length of K flow it will become as $\mathbf{z}^{(K)}$.

Then, $\mathbf{z}^{(K)}$ will be passed into the decoder network to extract hidden features. Similarly, the purpose of those hidden features is to derive the mean $\mu_{\mathbf{x}}$ and standard deviation $\sigma_{\mathbf{x}}$ of the input trace vector. After that, the reconstructed \mathbf{x} will be sampled from $\mathcal{N}(\mu_{\mathbf{x}}, \sigma_{\mathbf{x}}^2 \mathbf{I})$

C Inference

When a new trace is given, the log-likelihood value will be computed against the model to detect whether the trace is abnormal or not. If the trace \mathbf{x} is significantly different than the normal trace, the value of a trace $\log p_\theta(\mathbf{x})$ is noticeably smaller than the value of the normal traces. Instead of manually setting the threshold of anomaly, we follow the work from Liu et al. [39] and use Kernel Density Estimation (KDE) [22] to learn the distribution of the normal traces log-likelihood. Specifically, we adopt the p-value [57] approach and set the value as 0.001 to check if the probability of the log-likelihood value not following the learned distribution.

If the trace contains any unseen *call chain*, it will be regarded as abnormal. Training is a continuous process since the code evolves and the call paths keep changing over time. We use a sliding window of last 14 days of trace to keep our model up-to-date.

Artifact Appendix

Abstract

This artifact includes the script to utilize CRISP as we presented in the paper.

Scope

The artifact allows: top-down analysis, bottom-up analysis, and preprocessing for anomaly detection. It does not come with any traces to analyze and those traces need to be provided by the user.

Contents

It contains the implementation of CRISP with corresponding script to run the analysis.

Hosting

The artifact is available at <https://github.com/uber-research/CRISP> under **atc-2022** branch.

Requirements

- Python3.6 is recommended to run the anomaly detection. Otherwise, any python3 version should be fine.
- Git is also needed.
- "git clone https://github.com/NetManAIOps/TraceAnomaly.git" is required under the root directory in order to run anomaly detection.

Setup

- "python3.6 -m pip install -r requirements.txt" to install the dependency for CRISP.
- "python3.6 -m pip install -r TraceAnomaly/requirements.txt" to install the dependencies for TraceAnomaly.
- You may need to install protobuf if the requirements.txt doesn't work in TraceAnomaly by "python3.6 -m pip install protobuf==3.12.4".
- specify "TRACE_DIR" in "bottom-up.sh", "top-down.sh", and "preprocess.sh".

Top-down Analysis

Before running, you need to specify the input, output, serviceName, operationName, and processor number in `top-down.sh`. Make sure the output directory already existed. To run the analysis, simpling typing

```
mkdir top-down-result
bash top-down.sh
```

By default, the script will use all processors to run. You can change the processor number with `--parallelism` knob in `top-down.sh` script.

The result will be in `top-down-result` folder. It will represent Figure 2, Figure 11, and Figure 13 in the original paper. The number and shape won't be exactly the same given the trace and endpoints are different from the paper.

Specifically, flamegraph like Figure 2 is generated as `flame-graph-P50.cct.svg`, `flame-graph-P95.cct.svg`, and `flame-graph-P99.cct.svg`. `criticalPaths.html` is like the heatmap in Figure 11 and please open it in browser. The differential flamegraph like Figure 13 can be viewed in `flame-graph-P50vsP95.cct.svg`

Bottom-up Analysis

To use the artifact, run

```
bash bottom-up.sh
```

The figure will be generated under `result-bottom-up/` folder, which looks like Figure 13 ~ Figure 20 from the paper.

Anomaly Detection

Data Preprocessing run `bash preprocess.sh` to run generate the data for anomaly detection. Note each training, normal, and abnormal data needs to be parsed twice as it is shown in `preprocess.sh`. The reasons is that we need to know the total number of call path to generate the data.

Training Please refer to <https://github.com/NetManAIops/TraceAnomaly>.

Result Parsing Please go back to root directory when parsing the results. The trained model and the predicted results will be in `TraceAnomaly/webankdata/`.

To parse the results, run `python3.6 parse-rnvp.py -i path_to_rnvp_file`.



Whale: Efficient Giant Model Training over Heterogeneous GPUs

Xianyan Jia¹, Le Jiang¹, Ang Wang¹, Wencong Xiao¹, Ziji Shi^{1,2}, Jie Zhang¹, Xinyuan Li¹, Langshi Chen¹, Yong Li¹, Zhen Zheng¹, Xiaoyong Liu¹, Wei Lin¹

¹Alibaba Group ²National University of Singapore

Abstract

The scaling up of deep neural networks has been demonstrated to be effective in improving model quality, but also encompasses several training challenges in terms of training efficiency, programmability, and resource adaptability. We present Whale, a general and efficient distributed training framework for giant models. To support various parallel strategies and their hybrids, Whale generalizes the programming interface by defining two new primitives in the form of model annotations, allowing for incorporating user hints. The Whale runtime utilizes those annotations and performs graph optimizations to transform a local deep learning DAG graph for distributed multi-GPU execution. Whale further introduces a novel hardware-aware parallel strategy, which improves the performance of model training on heterogeneous GPUs in a balanced manner. Deployed in a production cluster with 512 GPUs, Whale successfully trains an industry-scale multimodal model with over ten trillion model parameters, named M6, demonstrating great scalability and efficiency.

1 Introduction

The training of large-scale deep learning (DL) models has been extensively adopted in various fields, including computer vision [15, 30], natural language understanding [8, 35, 43, 44], machine translation [17, 26], and others. The scale of the model parameters has increased from millions to trillions, significantly improving model quality [8, 24], but this has come at the cost of considerable efforts to efficiently distribute the model across GPUs. The commonly used data parallelism (DP) strategy is a poor fit, since it requires the model replicas in GPUs perform gradient synchronization proportional to the model parameter size for every mini-batch, thus easily becoming a bottleneck for giant models. Moreover, training trillions of model parameters requires terabytes of GPU memory at the minimum, which is far beyond the capacity of a single GPU.

To address the aforementioned challenges, a series of new parallel strategies in training DL models have been pro-

posed, including model parallelism (MP) [25], pipeline parallelism [20, 32], etc. For example, differing from the DP approach where each GPU maintains a model replica, MP partitions model parameters into multiple GPUs, avoiding gradient synchronization but instead letting tensors flow across GPUs.

Despite such advancements, new parallel strategies also introduce additional challenges. First, different components of a model might require different parallel strategies. Consider a large-scale image classification task with 100K classes, where the model is composed of *ResNet50* [19] for feature extraction and Fully-Connected (FC) layer for classification. The parameter size of *ResNet50* is 90 MB, and the parameter size of FC is 782 MB. If DP is applied to the whole model, the gradient synchronization of FC will become the bottleneck. One better solution is to apply DP to *ResNet50* and apply MP to FC (Section 2.3). As a result, the synchronization overhead can be reduced by 89.7%, thereby achieving better performance [25].

Additionally, using those advanced parallel strategies increases user efforts significantly. To apply DP in distributed model training, model developers only need to program the model for one GPU and annotate a few lines, and DL frameworks can replicate the execution plan among multiple GPUs automatically [27]. However, adopting advanced parallelism strategies might make different GPUs process different partitions of the model execution plan, which is difficult to achieve automatically and efficiently [23, 46]. Therefore, significant efforts are required for users to manually place computation operators, coordinate pipeline among mini-batches, implement equivalent distributed operators, and control computation-communication overlapping, etc. [26, 38, 41, 43]. Such an approach exposes low-level system abstractions and requires users to understand system implementation details when programming the models, which greatly increases the amount of user effort.

Further, the training of giant models requires huge computing resources. In industry, the scheduling of hundreds of homogeneous high-end GPUs usually requires a long queuing

time. Meanwhile, heterogeneous GPUs can be obtained much easier (e.g., a mixture of P100 [2] and V100 [3]) [21, 47]. But training with heterogeneous GPUs efficiently is even more difficult, since both the computing units and the memory capacity of GPUs need to be considered when building the model. In addition, due to the dynamic scheduling of GPUs, users are unaware of the hardware specification when building their models, which brings a gap between model development and the hardware environment.

We propose Whale, a deep learning framework designed for training giant models. Unlike the aforementioned approaches in which the efficient model partitions are searched automatically or low-level system abstractions and implementation details are exposed to users, we argue that deep learning frameworks should offer high-level abstractions properly to support complicated parallel strategies by utilizing user hints, especially when considering the usage of heterogeneous GPU resources. Guided by this principle, Whale strikes a balance by extending two necessary primitives on top of TensorFlow [7]. Through annotating a local DL model with those primitives, Whale supports all existing parallel strategies and their combinations, which is achieved by automatically rewriting the deep learning execution graph. This design choice decouples the parallel strategies from model code, and lowers them into dataflow graphs, which not only reduces user efforts but also enables graph optimizations and resources-aware optimizations for efficiency and scalability. In this way, Whale eases users from the complicated execution details of giant model training, such as scheduling parallel executions on multiple devices, and balancing computation workload among heterogeneous GPUs. Moreover, Whale introduces a hardware-aware load balancing algorithm when generating a distributed execution plan, which bridges the gap between model development and the heterogeneous runtime environment.

We summarize the key contributions of Whale as follows:

1. For carefully balancing user efforts and distributed graph optimization requirements, Whale introduces two new high-level primitives to express all existing parallel strategies as well as their hybrids.
2. By utilizing the annotations for graph optimization, Whale can transform local models into distributed models, and train them on multiple GPUs efficiently and automatically.
3. Whale proposes a hardware-aware load balancing algorithm, which is seamlessly integrated with parallel strategies to accelerate training on heterogeneous GPUs.
4. Whale demonstrates its capabilities by setting a new milestone in training the largest multi-modality pre-trained model M6 [28] with ten trillion model parameters, which requires only four lines of code change to scale the model and run on 512 NVIDIA V100M32 GPUs (Section 5.3.2).

Whale has been deployed as a production system for large-scale deep learning training at Alibaba. Using heterogeneous GPUs, further speedup of Bert-Large [13], Resnet50 [19], and GNMT [48] from 1.2x to 1.4x can be achieved owing to the hardware-aware load balancing algorithm in Whale. Whale also demonstrates its capabilities in the training of industry-scale models. With only four-line changes to a local model, Whale can train a Multi-Modality to Multi-Modality Multitask Mega-transformer model with 10 billion parameters (M6-10B) on 256 NVIDIA V100 GPUs (32GB), achieving 91% throughput in scalability. What's more, Whale scales to ten trillion parameters in model training of M6-10T using tensor model parallelism on 512 V100 GPUs (32GB), setting a new milestone in large-scale deep learning model training.

2 Background and Motivation

In this section, we first recap the background of distributed DL model training, especially the parallel strategies for large model training. We then present the importance and the challenges of utilizing heterogeneous GPU resources. Finally, we discuss the gaps and opportunities among existing approaches to motivate the design of a new training framework.

2.1 Parallel Strategies

Deep learning training often consists of millions of iterations, referred to as *mini-batches*. A typical mini-batch includes several phases to process data for model updating. Firstly, the training data is fed into the model layer-by-layer to calculate a set of scores, known as a *forward* pass. Secondly, a training loss is calculated between the produced scores and desired scores, which is then utilized to compute gradients for model parameters, referred to as a *backward* pass. Finally, the gradients scaled by a learning rate are used to update the model parameters and optimizer states.

Data parallelism. Scaling to multiple GPUs, data parallelism is a commonly adopted strategy where each worker holds a full model replica to process different training data independently. During the backward pass of every mini-batch, the gradients are averaged through worker synchronization. Therefore, the amount of communication is proportional to the model parameter size.

Pipeline Parallelism. As shown in Figure 1, a DL model is partitioned into two modules, *i.e.*, M0 and M1 (which are also named pipeline stages), which are placed on 2 GPUs respectively. The training data of a mini-batch is split into two smaller *micro-batches*. In particular, GPU0 starts with the forward of the 1st micro-batch on M0, and then it switches to process the forward of the 2nd micro-batch while sending

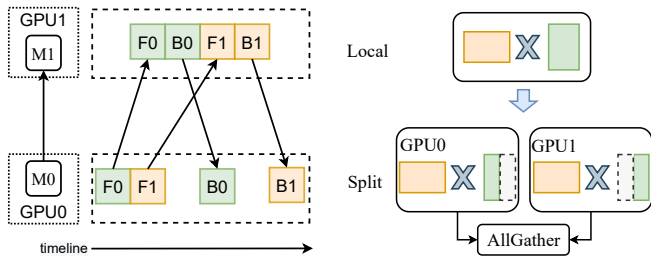


Figure 1: Pipeline parallelism of 2 micro-batches on 2 GPUs. Figure 2: Tensor model parallelism for *matmul* on 2 GPUs.

the output of the 1st micro-batch to GPU1. After GPU1 finishes processing forward and backward of the 1st micro-batch on M1, GPU0 continues to calculate the backward pass for M0 after receiving the backward output of M1 from GPU1. Therefore, micro-batches are pipelined among GPUs, which requires the runtime system to balance the load and overlap computation and communication carefully [16, 20, 32, 54]. The model parallelism [11, 12] can be treated as a special case of pipeline parallelism with only one micro-batch.

Tensor Model Parallelism. With the growing model size, to process DL operators beyond the memory capacity of the GPU, or to avoid significant communication overhead across model replicas, an operator (or several operators) might be split over multiple GPUs. The tensor model parallelism strategy partitions the input/output tensors and requires an equivalent distributed implementation for the corresponding operator. For example, Figure 2 illustrates the tensor model parallelism strategy for a *matmul* operator (*i.e.*, matrix multiplication) using 2 GPUs. A *matmul* operator can be replaced by two *matmul* operators, wherein each operator is responsible for half of the original computation. An extra all-gather operation is required to merge the distributed results.

In selecting a proper parallel strategy for model training, both model properties and resources need to be considered. For example, transformer [44] is an important model in natural language understanding, which can be trained efficiently using pipeline parallelism on a few GPUs (*e.g.*, 8 V100 GPUs with NVLINK [4]). However, pipeline parallelism does not scale well with more GPUs (*e.g.*, 64 V100 GPUs). Given more GPUs, each training worker is allocated with fewer operators, of which the GPU computation is not sufficient enough to overlap with the inter-worker communication cost, resulting in poor performance. Therefore, a better solution is to apply hybrid parallelism, where model partitions can be applied with different parallel strategies in combination, and parallel strategies can be nested. Particularly, for the training of a transformer model on 64 GPUs, the model parameters can be partitioned into 8 GPUs using a pipeline strategy, and apply model replica synchronization among 8 pipelined groups using nested data parallelism. Moreover, different parallel strategies can also apply to different model partitions for a hybrid.

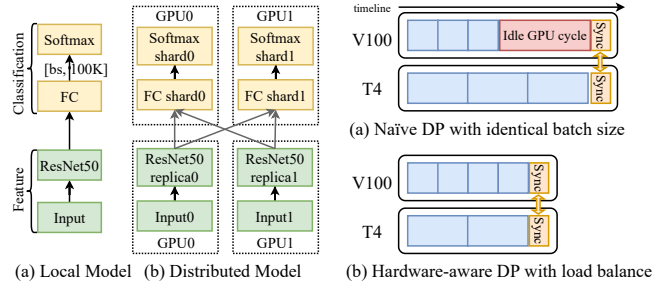


Figure 3: Hybrid parallelism for image classification. Figure 4: Data parallelism on heterogeneous GPUs

As an example, a large-scale image classification model (*i.e.*, 100K categories) consists of the image feature extraction partition and the classification partition. The image feature extraction partition requires a significant amount of computation on fewer model parameters. Conversely, the classification partition includes low-computation *fully-connected* and *softmax* layers, which are often 10x larger in model size compared to that of image feature extraction. Therefore, adopting a homogeneous parallel strategy will hinder the performance of either partitions. Figure 3 illustrates a better hybrid parallelism approach, in which data parallelism is applied for features extraction partition, tensor model parallelism is adopted for classification partition, and the two are connected.

2.2 Heterogeneity in GPU Clusters

Training a giant model is considerably resource-intensive [17, 33]. Moreover, distributed model training often requires resources to arrive at the same time (*i.e.*, gang schedule [21, 50]). In industry, the shared cluster for giant model training is usually mixed with various types of GPUs (*e.g.*, V100, P100, and T4) for both model training and inference [47]. Training giant models over heterogeneous GPUs lowers the difficulty of collecting all required GPUs (*e.g.*, hundreds or thousands of GPUs) simultaneously, therefore speeding up the model exploration and experiments. However, deep learning frameworks encounter challenges in efficiently utilizing heterogeneous resources. Different types of GPUs are different in terms of GPU memory capacity (*e.g.*, 16GB for P100 and 32GB for V100) and GPU computing capability, which natively introduces an imbalance in computational graph partition and deep learning operator allocation. Figure 4 illustrates training a model using data parallelism on two heterogeneous GPUs, *i.e.*, V100 and T4. The V100 training worker completes forward and backward faster when training samples are allocated evenly, thereby leaving idle GPU cycles before gradient synchronization at the end of every mini-batch. Through the awareness of hardware when dynamically generating an execution plan, Whale allocates more training samples (*i.e.*, batch-size=4) for V100 and the rest of 2 samples for T4 to eliminate the idle waiting time. Combined with advanced parallel strategies and the hybrids over heterogeneous GPUs, different GPU memory capacities and capabilities need to

be further considered when partitioning the model for efficient overlapping, which is a complex process (Section 3.3). Model developers can hardly consider all resources issues when programming, and we argue that developers should not have to. A better approach for a general deep learning framework would be automatically generating the execution plan for heterogeneous resources adaptively.

2.3 Gaps and Opportunities

Recent approaches [20, 26, 38, 41, 43] have been proposed for giant model training, however, with limitations as a general DL framework. Firstly, they only support a small number of parallel strategies, which lack a unified abstraction to support all of the parallel strategies and the hybrids thereof. Secondly, significant efforts are required in code modifications to utilize the advanced parallel strategies, compared with local model training and *DP* approach. Mesh-tensorflow [41] requires the re-implementation of DL operators in a distributed manner. Megatron [43], GPipe [20], DeepSpeed [38], and GShard [26] require user code refactoring using the exposed low-level system primitives or a deep understanding for the implementation of parallel strategies. Thirdly, automatically parallel strategy searching is time-consuming for giant models. Although Tofu [46] and SOAP [23] accomplish model partitioning and replication automatically through computational graph analysis, the search-based graph optimization approach has high computational complexity, which is further positively associated with the number of model operators (*e.g.*, hundreds of thousands of operators for GPT3 [8]) and allocated GPUs (*e.g.*, hundreds or thousands), making such an approach impractical when applying to giant model training. Finally, due to the heterogeneity in both GPU computing capability and memory, parallel strategies should be used adaptively and dynamically.

There are significant gaps in supporting giant model training using existing DL frameworks. Exposing low-level interfaces dramatically increases user burden and limits system optimization opportunities. Users need to understand the implementation details of distributed operators and handle the overlapping of computation with communication, which is hard for model developers. Using a low-level approach tightly couples model code to a specific parallel strategy, which requires code rewriting completely when switching between parallel strategies (*i.e.*, from pipeline parallelism to tensor model parallelism). More constraints are introduced to model algorithm innovations, because the efforts of implementing a new module correctly in hybrid strategies are not trivial, let alone consider the performance factors such as load balancing and overlapping. From the system aspect, seeking a better parallel strategy or a combination using existing ones also requires rewriting user code, demanding a deep understanding of the DL model.

To address the aforementioned challenges, Whale explores

a new approach that supports various parallel strategies while minimizing user code modifications. By introducing new unified primitives, users can focus on implementing the model algorithm itself, while switching among various parallel strategies by simply changing the annotations. Whale runtime utilizes the user annotations as hints to select parallel strategies at best effort with automatic graph optimization under a limited search scope. Whale further considers heterogeneous hardware capabilities using a balanced algorithm, making resource heterogeneity transparent to users.

3 Design

In this section, we first introduce key abstractions and parallel primitives which can express flexible parallelism strategies with easy programming API (Section 3.1). Then, we describe our parallel planner that transforms a local model with parallel primitives into a distributed model, through partitioning *TaskGraphs*, inserting bridge layers to connect hybrid strategies, and placing *TaskGraphs* on distributed devices (Section 3.2). In the end, we propose a hardware-aware load balance algorithm to speed up the training with heterogeneous GPU clusters (Section 3.3).

3.1 Abstraction

3.1.1 Internal Key Concepts

Deep learning frameworks such as TensorFlow [7] provide low-level APIs for distributed computing, but is short of abstractions to represent advanced parallel strategies such as pipeline. The lack of proper abstractions makes it challenging in the understanding and implementation of complicated strategies in a unified way. Additionally, placing model operations to physical devices properly is challenging for complicated hybrid parallel strategies, especially in heterogeneous GPU clusters. Whale introduces two internal key concepts, *i.e.*, *TaskGraph* and *VirtualDevice*. *TaskGraph* is used to modularize operations for applying a parallel strategy. *VirtualDevice* hides the complexity of mapping operations to physical devices. The two concepts are abstractions of internal system design and are not exposed to users.

TaskGraph(*TG*) is a subset of the model for parallel transformation and execution. One model can have one or more non-overlapping *TaskGraphs*. We can apply parallel strategies to each *TaskGraph*. By modularizing model operations into *TaskGraphs*, Whale can apply different strategies to different model parts, as well as scheduling the execution of *TaskGraphs* in a pipeline. A *TaskGraph* can be further replicated or partitioned. For example, in data parallelism, the whole model is a *TaskGraph*, which can be replicated to multiple devices. In pipeline parallelism, one pipeline stage is a *TaskGraph*. In tensor model parallelism, we can shard the *TaskGraph* into multiple submodules for parallelism.

```
import whale as wh
wh.init(wh.Config({
    "num_micro_batch": 8}))
with wh.replicate(1):
    model_stage1()
with wh.replicate(1):
    model_stage2()
```

Example 1: Pipeline with 2 TaskGraphs

```
import whale as wh
wh.init()
with wh.replicate(total_gpu):
    features = ResNet50(inputs)
with wh.split(total_gpu):
    logits = FC(features)
predictions = Softmax(logits)
```

Example 2: Hybrid of replicate and split

VirtualDevice (VD) is the logical representation of computing resources, with one *VirtualDevice* having one or more physical devices. *VirtualDevice* hides the complexity of device topology, computing capacity as well as device placement from users. One *VirtualDevice* is assigned to one *TaskGraph*. Different *VirtualDevices* are allowed to have different or the same physical devices. For example, VD0 contains physical devices GPU0 and GPU1, VD1 contains physical devices GPU2 and GPU3 (different from VD0), and VD2 contains physical devices GPU0 and GPU1 (the same as VD0).

3.1.2 Parallel Primitives

The parallel primitive is a Python context manager, where operations defined under it are modularized as one *TaskGraph*. Each parallel primitive has to be configured with a parameter *device_count*, which is used to generate a *VirtualDevice* by mapping the *device_count* number of physical devices. Whale allows users to suggest parallel strategies with two unified primitives, i.e., *replicate* and *split*. The two primitives can express all existing parallel strategies, as well as a hybrid of them [20, 25, 26, 32, 43].

replicate(device_count) annotates a *TaskGraph* to be replicated. *device_count* is the number of devices used to compute the *TaskGraph* replicas. If *device_count* is not set, Whale allocates a *TaskGraph* replica per device. If a *TaskGraph* is annotated with *replicate(2)*, it is replicated to 2 devices, with each *TaskGraph* replica consuming half of the mini-batch. Thus the mini-batch size for one model replica is kept unchanged.

split(device_count) annotates a *TaskGraph* to apply intra-tensor sharding. The *device_count* denotes the number of partitions to be sharded. Each sharded partition is placed on one device. For example, *split(2)* shards the *TaskGraph* into 2 partitions and placed on 2 devices respectively.

The parallel primitives can be used in combination to apply different parallel strategies to different partitions of the model. Additionally, Whale also provides JSON Config API to enable system optimizations. The config *auto_parallel* is used to enable automatic *TaskGraph* partitioning given a provided partition number *num_task_graph*, which further eases the programming for users and is necessary for hardware-aware optimization when resource allocation is dynamic (Section 3.3). In Whale, pipeline parallelism is viewed as an efficient inter-*TaskGraph* execution strategy. Whale uses the

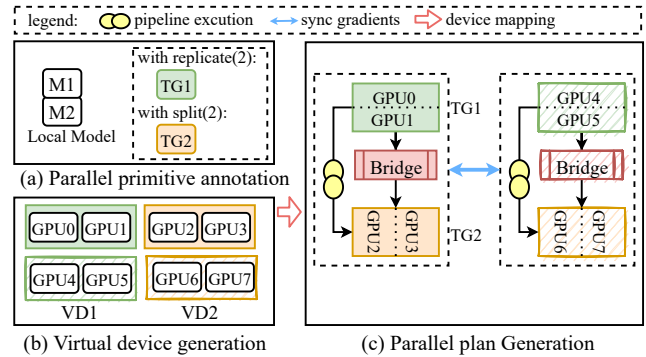


Figure 5: Whale Overview

config *num_micro_batch* to enable efficient pipeline parallelism among *TaskGraphs* when the value is greater than 1. In this way, Whale decouples the generation of *TaskGraph* from the choice of pipeline parallelism strategies [16, 20, 32]. The system can easily extend to incorporate more pipeline strategies (e.g., swap the execution order of *B0* and *F1* for *M1* in Figure 1).

Besides the combination of parallel strategies or pipeline parallelism, Whale further supports nested data parallelism to the whole parallelized model. Nested data parallelism is enabled automatically when the number of available devices is times of total devices requested by *TaskGraphs*.

Example 1 shows an example of pipeline parallelism with two *TaskGraphs*, with each *TaskGraph* being configured with 1 device. The pipeline parallelism is enabled by configuring the *pipeline.num_micro_batch* to 8. The total device number of the two *TaskGraphs* is summed to 2. If the available device number is 8, which is 4 times of total device number, Whale will apply a nested 4-degree data parallelism beyond the pipeline. In contrast, when using two available devices, it is a pure pipeline. Example 2 shows a hybrid strategy that replicates *ResNet50* feature part while splitting the *classification* model part for the example in Figure 3.

```
wh.init(wh.Config({"num_task_graph": 2,
    "num_micro_batch": 4, "auto_parallel": True}))
model_def()
```

Example 3: Auto pipeline

Example 3 shows an automatic pipeline example with two *TaskGraphs*. When *auto_parallel* is enabled, Whale will partition the model into *TaskGraphs* automatically according to the computing resource capacity and the model structure. (Section 3.3)

3.2 Parallel Planner

The parallel planner is responsible for producing an efficient parallel execution plan, which is the core of Whale runtime. Figure 5 shows an overview of the parallel planner. The workflow can be described as follows: (a) The parallel planner takes a local model with optional user annotations, computing

resources, and optional configs as inputs. The model hyperparameters (e.g., batch size and learning rate), and computing resources (e.g., #GPU and #worker) are decided by the users manually, while the parallel primitive annotations and configs (e.g., `num_task_graph` and `num_micro_batch`) could be either be manual or decided by Whale automatically; (b) the *VirtualDevices* are generated given computing resources and optional annotations automatically (Section 3.2.1); and (c) the model is partitioned into *TaskGraphs*, and the *TaskGraph* is further partitioned internally if `split` is annotated. Since we allow applying different strategies to different *TaskGraphs*, there may exist an input/output mismatch among *TaskGraphs*. In such case, the planner will insert the corresponding bridge layer automatically between two *TaskGraphs* (Section 3.2.3).

3.2.1 Virtual Device Generation

VirtualDevices are generated given the number of devices required by each *TaskGraph*. Given K requested physical devices $GPU_0, GPU_1, \dots, GPU_K$ and a model with N *TaskGraphs*, with corresponding device number d_1, d_2, \dots, d_N . For the i^{th} *TaskGraph*, Whale will generate a *VirtualDevice* with d_i number of physical devices. The physical devices are taken sequentially for each *VirtualDevice*. As mentioned in Section 3.1.2, when the available device number K is divisible by the total number of devices requested by all *TaskGraphs* $\sum_i^N d_i$, Whale will apply a nested *DP* of $\frac{K}{\sum_i^N d_i}$ -degree to the whole model. In such case, we also replicate the corresponding *VirtualDevice* for *TaskGraph* replica. By default, devices are not shared among *TaskGraphs*. Sharing can be enabled to improve training performance in certain model sharding cases by setting cluster configuration¹. Whale prefers to place one model replica (with one or more *TaskGraphs*) within a node, and replicates the model replicas across nodes. Advanced behaviors such as placing *TaskGraph* replicas within a node to utilize NVLINK for AllReduce communication can be achieved by setting the aforementioned configuration. For example, as shown in Figure 5, there are two *TaskGraphs*, and each *TaskGraph* requests 2 GPUs. Two *VirtualDevices* VD1 and VD2 are generated for two *TaskGraphs*. VD1 contains GPU_0 and GPU_1 , and VD2 contains GPU_2 and GPU_3 . As the number of available GPUs is 8, which is divisible by the total GPU number of *TaskGraphs* 4, a replica of *VirtualDevices* can be generated but with different physical devices.

3.2.2 TaskGraph Partitioning

Whale first partitions a model into *TaskGraphs*, either by using explicit annotations or automatic system partitioning. If a user annotation is given, operations defined within certain parallel primitive annotation compose a *TaskGraph*. Otherwise, the system generates *TaskGraphs* based on the given config

¹<https://easyparallellibrary.readthedocs.io/en/latest/api/config.html#clusterconfiguration>

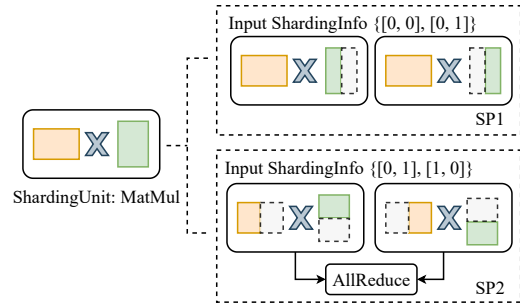


Figure 6: Sharding pattern example for MatMul. One ShardingUnit can map to multiple sharding patterns.

parameter `num_task_graph` and hardware information. The details of the hardware-aware model partitioning is described in Section 3.3.

If a *TaskGraph* is annotated with `split(k)`, Whale will automatically partition it by matching and replacing sharding patterns with a distributed implementation. Before describing the sharding pattern, we introduce two terminologies for tensor model parallelism: 1) *ShardingUnit* is a basic unit for sharding, and can be an operation or a layer with multiple operations; and 2) *ShardingInfo* is the tensor sharding information, and is represented as a list $[s_0, s_1, \dots, s_n]$ given a tensor with n dimensions, where s_i represents whether to split the i^{th} dimension, 1 means true and 0 means false. For example, given a tensor with shape $[6, 4]$, the *ShardingInfo* $[0, 1]$ indicates splitting in the second tensor dimension, whereas $[1, 1]$ indicates splitting in both dimensions. A sharding pattern (SP) is a mapping from a *ShardingUnit* and input *ShardingInfo* to its distributed implementations. For example, Figure 6 shows two sharding patterns SP1 and SP2 with different input *ShardingInfo* for *ShardingUnit* MatMul.

To partition the *TaskGraph*, Whale first groups the operations in the split *TaskGraph* into multiple *ShardingUnits* by hooking TensorFlow ops API². The *TaskGraph* sharding process starts by matching *ShardingUnits* to the predefined sharding patterns in a topology order. A pattern is matched by a *ShardingUnit* and input *ShardingInfos*. If multiple patterns are matched, the pattern with a smaller communication cost is selected. Whale replaces the matched pattern of the original *ShardingUnit* with its distributed implementation.

3.2.3 Bridge Layer

When applying different parallel strategies to different *TaskGraphs*, the input/output tensor number and shape may change due to different parallelism degrees or different parallel strategies, thereby resulting in a mismatch of input/output tensor shapes among *TaskGraphs*. To address the mismatch, Whale proposes a *bridge layer* to gather the distributed tensors and feed them to the next *TaskGraph*.

²TensorFlow Ops: <https://github.com/tensorflow/tensorflow/tree/r1.15/tensorflow/python/ops>

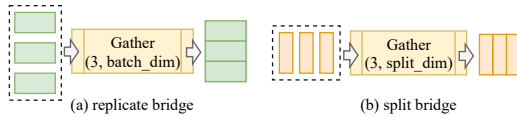


Figure 7: Bridge patterns.

Whale designs two *bridge patterns* for *replicate* and *split* respectively, as shown in Figure 7. For *replicate*, the *TaskGraph* is replicated to N devices, with different input batches. The *bridge layer* gathers the outputs from different batches for concatenation in batch dimension $batch_dim$. For *split*, the outputs of *TaskGraph* are partitioned in split dimension $split_dim$. The *bridge layer* gathers *TaskGraph* outputs for concatenation in $split_dim$. By using the *bridge layer*, each *TaskGraph* can obtain a complete input tensor. If the gather dimension of the *bridge layer* is the same as the successor *TaskGraph* input partition dimension, Whale will optimize by fusing the aforementioned two operations to reduce the communication overhead. As an example, if the outputs of the *TaskGraph* are gathered in the first dimension, and the inputs of the successor *TaskGraph* are partitioned in the same dimension, then Whale will remove the above *gather* and *partition* operations.

3.3 Hardware-aware Load Balance

In this section, we describe how we utilize the hardware information to balance the workloads among *TaskGraphs*, which achieves high performance even in heterogeneous GPU clusters. The Whale parallel planner obtains the hardware information from the cluster scheduler when the training job is launched, and is responsible for both intra-*TaskGraph* and inter-*TaskGraph* load balancing.

3.3.1 Intra-TaskGraph Load Balance

When the allocated devices are homogeneous, by default Whale distributes the workloads within a *TaskGraph* to multiple devices evenly. However, when allocated with heterogeneous GPUs with different computing capacities (e.g., V100 and P100), the aforementioned identical distribution effectuates suboptimal performance. Such performance can be attributed to a synchronization barrier at the end of *TaskGraph* execution, which leads to long idle GPU time for the faster GPU, as shown in Figure 4(a). To improve the overall utilization of heterogeneous GPUs, we need to balance the computing according to the device’s computing capacity. The intra-*TaskGraph* load balance attempts to minimize the idle time within a *TaskGraph*, which is achieved by balancing the workloads proportional to device computing capacity while being subject to memory constraints. For a *TaskGraph* annotated with *replicate*, Whale balances the workload by adjusting the batch size for each *TaskGraph* replica. The local batch size on heterogeneous devices might differ due to the load balancing strategy (Whale keeps the global batch size

unchanged). If batch-sensitive operators such as *BatchNorm* exist, the local batch differences might have statistical effects. Yet, no users suffered convergence issues in our experiments or in our production deployment, which is probably due to the robustness of *DL*. Besides, techniques like SyncBatch-Normalization³ might help. For a *TaskGraph* annotated with *split*, Whale balances the FLOP of a partitioned operation through uneven sharding in splitting dimension among multiple devices.

We profile the *TaskGraph TG* on single-precision floating-point operations(FLOP) as TG_{flop} and peak memory consumption as TG_{mem} . Given N GPUs, we collect the information for device i including the single-precision FLOP per second as DF_i and memory capacity as DM_i . Assuming the partitioned load ratio on the device i is L_i , we need to find a solution that minimizes the overall GPU waste, which is formulated in Formula 1. We try to minimize the ratio of the computational load of the actual model for each device L_i and the ratio of the computing capacity of the device over the total cluster computing capacity $DF_i / \sum_{i=0}^N DF_i$, the maximum workload being bounded by the device memory capacity DM_i .

$$\min \sum_i^N \left\| L_i - \frac{DF_i}{\sum_{i=0}^N DF_i} \right\| \quad (1)$$

$$\text{s.t. } \sum_{i=0}^N L_i = 1; L_i * TG_{mem} \leq DM_i, (i = 1, 2, \dots, N)$$

The load ratio L_i in each device is initialized in proportional to the device’s computing capacity, which ideally results in a most balanced partition. However, when the memory constraint is not satisfied, we need to adjust the load allocation to avoid out-of-memory (OOM) errors, while still trying to achieve good performance. Whale proposes a memory-constraint balancing algorithm to balance the workloads among devices. The main idea of the algorithm is to shift the workload from the memory-overload device to a memory-free device with the lowest computation load. The details of the algorithm are illustrated in Algorithm 1. It takes a *TaskGraph TG* and *VirtualDevice* with N physical devices as inputs. The algorithm first initializes (line 3-10) the profiling results including 1) *load_ratios* as the workload ratios of devices; 2) *mem_utils* as the memory utilization of devices; 3) *flop_utils* as the FLOP utilization of devices; 4) *oom_devices* records out of memory devices whose value in *mem_utils* is greater than 1; and 5) *free_devices* records devices that have free memory space. The algorithm then iteratively shifts the load from a memory-overload device to a memory-available device (line 11-18). It first finds a *peak_device* with maximum memory utilization from *oom_devices*, then it finds a *valley_device* with available memory space and the lowest

³https://www.tensorflow.org/api_docs/python/tf/keras/layers/experimental/SyncBatchNormalization

Algorithm 1: Memory-Constraint Load Balancing

Input: $TaskGraph\ TG, VirtualDevice(N)$

```
1 load_ratios = 0; mem_utils = 0; flop_utils = 0
2 oom_devices = 0; free_devices = 0
3 foreach  $i \in 0..N$  do
4   load_ratios[i] =  $\frac{DF_i}{\sum_{i=0}^N DF_i}$ 
5   mem_utils[i] =  $\frac{load\_ratios[i]*TG_{mem}}{DM_i}$ 
6   flop_utils[i] =  $\frac{load\_ratios[i]*TG_{flop}}{DF_i}$ 
7   if mem_utils[i] > 1 then
8     oom_devices.append(i)
9   else
10    free_devices.append(i)
11 while oom_devices  $\neq$  0 & free_devices  $\neq$  0 do
12   peak_device = argmax(oom_devices, key = mem_utils)
13   valley_device = argmin(free_devices, key =
14     (flop_utils, mem_utils))
15   if shift_load(peak_device, valley_device) == success
16     then
17     update_profile(mem_utils, flop_utils)
18     oom_devices.pop(peak_device)
19   else
20     free_devices.pop(valley_device)
```

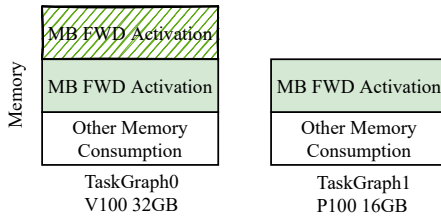


Figure 8: Pipeline *TaskGraphs* on heterogeneous GPUs

FLOP utilization. The *shift_load* function attempts to shift the workload from a *peak_device* to a *valley_device*. For data parallelism, the batch size in the *peak_device* is decreased by b , and the batch size in the *valley_device* is increased by b . b is the maximum number that the *valley_device* will not go OOM after getting the load from the *peak_device*. The profiling information for each device is updated after a successful workload shift is found. The aforementioned process iterates until the *oom_devices* are empty or the *free_devices* are empty.

3.3.2 Inter-TaskGraph Load Balance

When multiple *TaskGraphs* are executed in a pipeline, we need to balance the inter-*TaskGraph* workloads on heterogeneous GPUs. As we introduced in Section 2.1, pipeline parallelism achieves efficient execution by interleaving forward/backward execution among multiple micro-batches. For a model with N *TaskGraphs*, the i^{th} *TaskGraph* needs to cache $N - i$ forward activations [32]. Notably, i^{th} *TaskGraph* has to

cache one more micro-batch forward activation than the previous *TaskGraph*. Since activation memory is proportional to batch size and often takes a large proportion of the peak memory, e.g., the activation memory VGG16 model with batch size 256 takes up around 74% of the peak memory [18], resulting in uneven memory consumption among different *TaskGraphs*. The different memory requirements of *TaskGraphs* motivate us to place earlier *TaskGraphs* on devices with higher memory capacity. This can be achieved by sorting and reordering the devices in the corresponding *VirtualDevice* by memory capacity, from higher to lower. Figure 8 shows the memory breakdown of the pipeline example (Figure 1) with two *TaskGraphs* over heterogeneous GPUs V100 (32GB) and P100 (16GB), we prefer putting *TaskGraph0* to V100, which has a higher memory config. The *TaskGraph* placement heuristic is efficient for common Transformer-based models (i.e., BertLarge and T5 in Figure 18). There might be cases where later stages contain large layers (i.e., large sparse embedding), which can be addressed in Algorithm 1 on handling OOM errors. After reordering the virtual device according to memory requirement, we partition the model operations to *TaskGraphs* in a topological sort and apply Algorithm 1 to balance the computing FLOP among operations, subject to the memory bound of the memory capacity of each device.

4 Implementation

Whale is implemented as a standalone library without modification of the deep learning framework, which is compatible with TensorFlow 1.12 and TensorFlow 1.15 [7]. The source code of Whale includes 13179 lines of Python code and 1037 lines of C++ code. We have open-sourced⁴ the Whale framework to help giant model training accessible to more users.

Whale enriches the local model with augmented information such as phase information, parallelism annotation, etc., which is crucial to parallelism implementation. To assist the analysis of the user model without modifying the user code, Whale inspects and overwrites TensorFlow build-in functions to capture augmented information. For example, operations are marked as backward when *tf.gradients* or *compute_gradients* functions are called.

The parallel strategy is implemented by rewriting the computation graph. We implement a general graph editor module for ease of graph rewriting, which includes functions such as subgraph clone, node replacement, dependency control, and so on. To implement data parallelism, Whale first clones all operations and tensors defined in a local *TaskGraph* and replaces the device for model replicas. Then it inserts NCCL [6] AllReduce [40] operation to synchronize gradients for each *TaskGraph* replica. To implement tensor model parallelism, Whale shards the *TaskGraph* by matching a series of predefined patterns, replacing them with corresponding distributed

⁴<https://github.com/alibaba/EasyParallelLibrary>

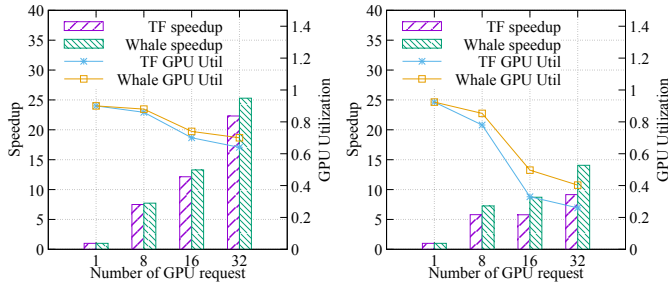


Figure 9: Whale DP vs TF DP on ResNet.

implementation, and inserting communication operations as needed. To implement pipeline parallelism, Whale builds a pipeline strategy module that supports state-of-the-art strategies [16, 20, 32]. By default, Whale adopts a backward-first strategy which is similar to PipeDream [32]. The pipeline strategy is implemented by first partitioning the minibatch into micro-batches. The interleaving of forward-backward micro-batch execution is achieved by inserting control dependency operations among entrance and exit operations of different *TaskGraphs*.

To assist hardware-aware optimizations, Whale implements profiling tools that profile the model FLOPS and peak memory consumption. The parallel planner gets the hardware information from our internal GPU cluster, which is used to generate an efficient parallel plan by balancing the computing workloads over heterogeneous GPUs.

Besides, Whale is highly optimized in both computing efficiency and memory utilization by integrating with a series of optimization technologies such as ZERO [36], recomputation [10], CPU offload [39], automatic mixed precision [31], communication optimization [40], XLA [7], etc.

5 Experiment

In this section, we first demonstrate the efficiency of the parallelism strategy by evaluating micro-benchmarks. We then evaluate the training with heterogeneous GPUs to show the advantages of the hardware-aware load balance algorithm. We end by showing the effectiveness and efficiency of Whale by two industry-scale multimodal model training cases. All the experiments are conducted on a shared cloud GPU cluster. Every cluster node is equipped with a 96-core Intel Xeon Platinum 8163 (Skylake) @2.50GHz with 736GB RAM, running CentOS 7.7. Each node consists of 2/4/8 GPUs, with NVIDIA 32-GB V100 GPUs [3] or NVIDIA 16-GB P100 GPUs [2], powered by NVIDIA driver 418.87, CUDA 10.0, and cuDNN 7. Nodes are connected by 50Gb/s ethernet. All the models are implemented based on TensorFlow 1.12.

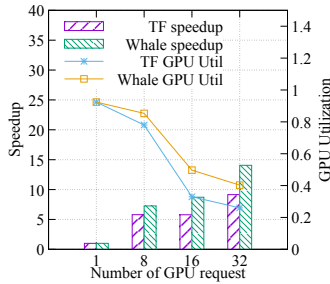


Figure 10: Whale DP vs TF DP on BertLarge.

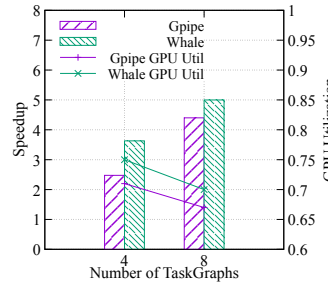


Figure 11: Whale Pipeline vs GPipe.

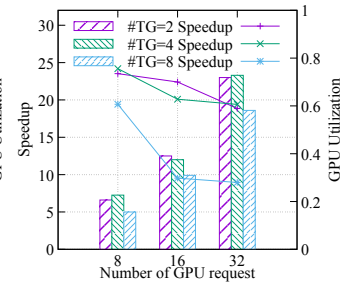


Figure 12: Hybrid pipeline parallelism on BertLarge.

5.1 Micro-benchmark

In this section, we evaluate Whale with a series of micro-benchmarks. We first demonstrate that Whale is efficient in single parallel strategy by comparing with TensorFlow Estimator [14] DP and GPipe [20] pipeline. We then show the advantages of Whale hybrid strategies over single parallel strategy. Next, we measure the overhead of the bridge layer for hybrid strategies. Finally, we evaluate the effect of sharding patterns in automatic *TaskGraph* partitioning.

5.1.1 Performance of Single Parallel Strategy

We evaluate Whale DP by comparing it with TensorFlow Estimator DP, using the BertLarge [13] and ResNet50 [19] on different number of V100 GPUs. Figure 9 and Figure 10 show the training throughput speedup on ResNet50 and BertLarge respectively. The throughput speedup is calculated by dividing the training throughput on N devices by the throughput on one device. Whale DP consistently obtained better speedup and higher GPU utilization than TensorFlow Estimator DP. Such findings could be attributed to Whale's communication optimization technologies such as hierarchical and grouped AllReduce, which is similar to Horovod [40].

We then evaluate the efficiency of Whale pipeline parallelism by comparing with GPipe [20]. The pipeline scheduling strategy in Whale is similar to PipeDream [32]. The experiments are conducted using the BertLarge model with 4/8 pipeline stages on the different numbers of V100 GPUs. As shown in Figure 11, the training throughput speedup of Whale outperforms GPipe in both 4 stages and 8 stages by 1.45X and 1.14X respectively. We attribute the performance gain to the use of the alternating forward-backward scheduling policy [32], which improves GPU utilization. We also find that the pipeline performance is sensitive to the *num_task_graph*, thus exposing it as a configurable parameter can help achieve a better performance when models and computing resources change.

5.1.2 Performance of Hybrid Strategy

We evaluate hybrid strategies by comparing them with the single parallel strategy. We also compare the performances of

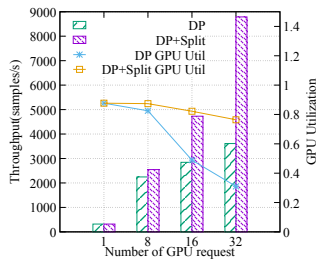


Figure 13: DP vs Hybrid on ResNet50 w/ 100K classes.

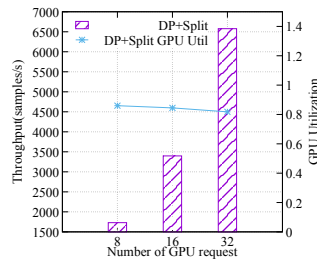


Figure 14: Hybrid strategy on ResNet50 w/ 1M classes.

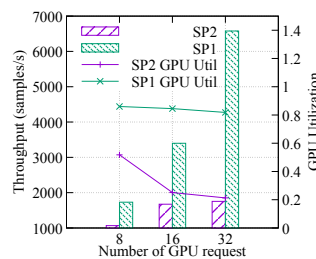


Figure 15: Effect of Sharding Pattern.

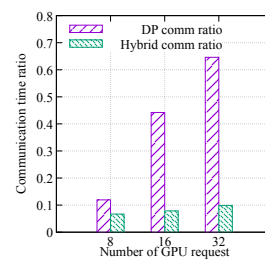


Figure 16: Overhead of Bridge Layer.

hybrid strategies on different numbers of devices. We select two typical types of hybrid strategies: 1) Nested pipeline with DP; and 2) Combination of DP and tensor model parallelism.

We first apply a nested pipeline with DP to the BertLarge model on V100 GPUs. The model is partitioned into 2/4/8 number of *TaskGraphs*, and we measure the training performance of each model on 8/16/32 GPUs. Figure 12 shows that pipelines with 2 *TaskGraphs* and 4 *TaskGraphs* get similar training speedups and GPU utilization. However, we observe a performance drop on 8 *TaskGraphs* and lower GPU utilization compared to 2/4 *TaskGraphs*. This is because 8 *TaskGraphs* lead to relatively fewer model operations in each *TaskGraph*, and the GPU computation is not enough to overlap the inter-*TaskGraph* communication, resulting in poor performance.

Next, we evaluate the combination hybrid strategy on a large-scale image classification model, as we have discussed in Section 2.1 and illustrated in Figure 3. We perform experiments on classification numbers 100K and 1M on different numbers of V100 GPUs. To reduce the communication overhead of hybrid parallelism, we collocate the *ResNet50* replicas with *FC* partitions. We compare the hybrid results of 100K classes with DP, as shown in Figure 13, hybrid parallelism outperforms data parallelism by 1.13X, 1.66X, and 2.43X training throughput speedup with 8, 16, and 32 GPUs respectively, with the line plot corresponding to GPU utilization. When the number of workers increases, hybrid parallelism maintains a near-linear speedup, while the DP strategy fails drastically beyond 16 workers. This is because the heavy *FC* layer (the parameter size of ResNet50 backbone is 90 MB, while the parameter size of *FC* layer is 782MB) incurs a huge gradient synchronization overhead. For the task of 1M classes, DP fails due to OOM. With hybrid parallelism, Whale allows for the training of image classification task with one million classes. Figure 14 shows the performance of hybrid parallelism over 8/16/32 GPUs. The training throughputs from 8 GPUs to 32 GPUs achieve 95% scaling efficiency, which highlights the need for using a hybrid strategy.

5.1.3 Overhead of Bridge Layer

To demonstrate the efficiency of the hybrid strategy, We measure the overhead of the bridge layer by profiling the bridge layer time with 100K classes on 8/16/32 GPUs. We then compare the overhead of gradient AllReduce time in DP with the

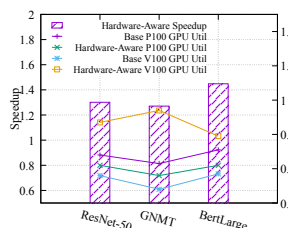


Figure 17: Hardware-Aware Data Parallelism.

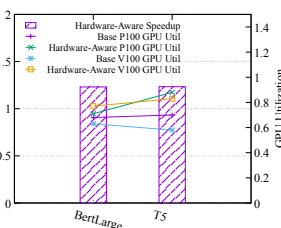


Figure 18: Hardware-Aware Pipeline Parallelism.

bridge overhead to understand the performance gain from hybrids. As shown in Figure 16, the overhead of the bridge layer takes around 6% in overall training time in 8 GPUs and 10% in 32 GPUs. The overhead of the hybrid is reduced by 6X on 32 GPUs compared to gradient synchronization overhead of pure DP.

5.1.4 Effect of Sharding Pattern

As Whale automatically chooses a sharding pattern with minimum communication cost (Section 3.2.2), to demonstrate the effect of exploring the sharding patterns, we force the framework to use a specific pattern in this experiment. We evaluate two types of sharding patterns as illustrated in Figure 6 on large scale image task with 100K classes. *SP1* shards the second input tensor in the second tensor dimension, and *SP2* shards the two input tensors and aggregates the results with *AllReduce*. The comparison results of the two sharding patterns are shown in Figure 15, where *SP1* outperforms *SP2* by 1.6X to 3.75X as the number of requested GPUs increases from 8 to 32, as *SP1* has a lower communication cost than *SP2*. The exploration of sharding patterns allows for the possibility of system optimization in distributed model implementation.

5.2 Performance of Load Balance

We show the benefits of the hardware-aware load balancing algorithm by evaluating data parallelism and pipeline parallelism.

For data parallelism, we evaluate three typical models, including ResNet50, BertLarge, and GNMT [48]. The experiments are conducted on heterogeneous GPUs that consist of 8 32GB V100 GPUs and 8 16GB P100 GPUs. We set the same batch size for all model replicas as the baseline. We


```

import whale as wh
wh.init(wh.Config({
    "num_micro_batch": 35,
    "num_task_graph": 8}))
# Define M6 model.
m6_model_def()

```

Example 4: M6-10B model with pipeline

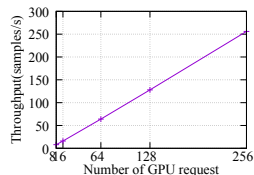


Figure 19: M6-10B with Pipeline and DP.

then apply the hardware-aware algorithm to each model and get the speedup compared with the baseline performance, as shown in Figure 17. Whale outperforms the baseline in all three models by a factor from 1.3X to 1.4X. We also measure GPU utilization and report the average metric for each GPU type. The hardware-aware policy significantly improves the GPU utilization of V100 by 1.39X to 1.96X for the three models, which improves the overall training performance.

For pipeline parallelism, we evaluate two models, including BertLarge and T5-Large [52]. The training is performed on heterogeneous GPUs that consist of 4 32GB V100 GPUs and 4 16GB P100 GPUs. Both BertLarge and T5-Large are partitioned into 4 stages. We further apply a nested DP to *pipeline*. We set the evenly partitioned model as the baseline. We conducted training with the hardware-aware policy and got about 20% speedup on both models, as shown in Figure 18. The GPU utilization of hardware-aware load balancing strategy improved the GPU utilization of V100 by around 40%, which shows the efficiency of the hardware-aware load balancing algorithm.

5.3 Industry-Scale Giant Model Training

5.3.1 Training M6-10B Model

The M6-10B [28] model is a Chinese multimodal model with 10 billion parameters. The model consists of 24 encoder layers and 24 decoder layers. We use Adafactor [42] as the training optimizer. We parallelize the training of M6-10B model with a hybrid parallel strategy, by nesting pipeline parallelism and data parallelism. Whale can easily scale a local M6 model to a distributed one by only adding a few lines on top of the model definition as shown in Example 4. We set the number of pipeline stages to 8 and the number of micro-batches to 35. We enable recomputation [10] to save activation memory during training. The training performance is evaluated on 32-GB V100 GPUs. Each node contains 8 GPUs. When scaling the computing nodes from 8 to 32, Whale achieved 91% scalability, as shown in Figure 19.

5.3.2 Training M6-MoE Model to Trillions

We scale the model parameters to 10 trillion (10T) by switching to hybrids of *DP* and tensor model parallelism with only a small number of lines of code change. The computation cost of training dense models is proportional to the model parameters. If we scale the dense 10B model to the dense

10T model linearly without considering overhead, we need at least 256,000 NVIDIA V100 GPUs. Instead of scaling the M6 model with dense structure, we adopt M6-MoE [53] model with sparse expert solution [17, 26]. The sample code of the MoE structure is implemented with Whale by adding four lines, as shown in Example 5. Line 3 sets the default parallel primitive as *replicate*, i.e., data parallelism is applied for the operations if not explicitly annotated. Line 5 partitions the computation defined under *split* scope across devices.

```

1 import whale as wh
2 wh.init()
3 wh.set_default_strategy(wh.replicate(total_gpus))
4 combined_weights, dispatch_inputs=gating_dispatch()
5 with wh.split(total_gpus):
6     outputs = MoE(combined_weights, dispatch_inputs)

```

Example 5: Distributed MoE model

We evaluate M6-MoE model with 100 billion, 1 trillion and 10 trillion parameters respectively, the detailed configurations can be found in [29, 53]. We enable built-in technologies of Whale to optimize the training process, such as recomputation [10], AMP (auto mixed precision) [1], XLA [5], CPU offloading [39], etc. We can train the M6-MoE-100B model with 100 million samples on 128 V100 in 1.5 days. We advance the model scale to 1 trillion parameters on solely 480 NVIDIA V100 GPUs, in comparison with the recent SOTA on 2048 TPU cores [17]. We further scale the model to 10 trillion parameters by adopting optimized tensor offloading strategies [29] with 512 NVIDIA V100 GPUs. Whale can scale models from 100 billion to 10 trillion without code changes, which makes giant model training accessible to most users.

6 Related Work

Giant model training. TensorFlow [7] and PyTorch [34] provide well-supported data parallelism and vanilla model parallelism by explicitly assigning operations to specific devices. However, they are not efficient enough for giant model training. Megatron [43], GPipe [20], and Dapple [16] have proposed new parallel training strategies to scale the training of large-scale models. DeepSpeed [38] lacks general support for tensor model parallelism, besides, model layers are required to rewrite in sequential for pipeline parallelism. GShard [26] supports operator splitting by introducing model weight annotations and tensor dimension specifications. The high performance of those works is achieved by exposing low-level system abstractions to users (e.g., device placement, equivalent distributed implementation for operators), or enforcing model or tensor partition manually, which results in significant user efforts. As a parallel work to Whale, GSPMD [51] extends GShard by annotating tensor dimensions mapping for both automatic and manual operator partitioning. As a general giant model training framework, Whale adopts a unified abstraction to express different parallel strategies and their hy-

brid nests and combinations, utilizing high-level annotations and pattern matching for operator splitting. Whale further scales to M6-10T through automatically distributed graph optimizations with the awareness of heterogeneous resources.

Zero [36, 37, 39] optimizes memory usage by removing redundant GPU memory, offloading computation to the CPU host, and utilizing non-volatile memory respectively. Recomputation [10] trades computation for memory by recomputing tensors from checkpoints. Such memory optimization approaches are orthogonal to Whale, which can be further combined for giant model training efficiently.

Graph optimization. Deep learning is powered by dataflow graphs with optimizations to rewrite the graph for better performance, such as TensorFlow XLA [7], TVM [9], Ansor [55], AStitish [56], *etc.* TASO [22] and PET [45] adopt a graph substitution approach to optimize the computation graph automatically. Those works mainly focus on the performance of a single GPU, while Whale utilizes the graph optimization approach for achieving efficient performance in distributed training. Tofu [46] and SOAP [23] also use graph partition to produce distributed execution plans, but with a high search cost. Whale utilizes the introduced annotations to shrink the search space, thus making graph optimization practical for giant model training at a trillion scale. Besides, Whale extends the graph optimization approach to complicated parallel strategies in a unified abstraction, capable of pipeline parallelism, tensor model parallelism, and hybrid parallelism.

Resource heterogeneity. Philly [21] reports the trace study in multi-tenant GPU clusters of Microsoft and shows the effect of gang scheduling on job queuing. MLaaS [47] studies a two-month trace of a heterogeneous GPU cluster in Alibaba PAI. Gandiva [49] shows jobs are different in sensitivity to allocated resources. Whale is capable of adapting to resource heterogeneity, which can reduce the queuing delay of giant model training with hundreds of GPUs. The design of Whale advocates the approach of decoupling model programming and distributed execution. It dynamically generates an efficient execution plan by considering the properties of both model and heterogeneous resources.

7 Conclusion

Whale demonstrates the possibility of achieving efficiency, programmability, and adaptability in a scalable deep learning framework for training trillion-parameter models. Whale supports various parallel strategies using a unified abstraction, hides distributed execution details through new primitive annotations, and adapts to heterogeneous GPUs with automatic graph optimizations. Going forward, we hope that Whale can become a large-scale deep learning training foundation to further engage model algorithm innovations and system optimizations in parallel, making giant model training technology to be adopted easily and efficiently at scale.

Acknowledgements

We would like to thank our anonymous shepherd and reviewers for their valuable comments and suggestions. We would also like to thank the M6 team and all users of Whale for their help and suggestions.

References

- [1] Automatic mixed precision for deep learning. <https://developer.nvidia.com/automatic-mixed-precision>.
- [2] Nvidia tesla p100. <https://www.nvidia.com/en-us/data-center/tesla-p100/>.
- [3] Nvidia v100 tensor core gpu. <https://www.nvidia.com/en-us/data-center/v100/>.
- [4] NVLink. <https://www.nvidia.com/en-us/data-center/nvlink/>.
- [5] Xla: Optimizing compiler for machine learning. <https://www.tensorflow.org/xla>.
- [6] Nccl. <https://developer.nvidia.com/nccl>, 2019.
- [7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.
- [8] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Nee-lakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.
- [9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.
- [10] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.
- [11] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pages 571–582, Broomfield, CO, October 2014. USENIX Association.

- [12] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc’Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.
- [13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [14] Joshua V Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A Saurous. Tensorflow distributions. *arXiv preprint arXiv:1711.10604*, 2017.
- [15] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.
- [16] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. Dapple: A pipelined data parallel approach for training large models, 2020.
- [17] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2021.
- [18] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. Estimating gpu memory consumption of deep learning models. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1342–1352, 2020.
- [19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965*, 2018.
- [21] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.
- [22] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.
- [23] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In Ameet Talwalkar, Virginia Smith, and Matei Zaharia, editors, *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org, 2019.
- [24] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.
- [25] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.
- [26] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.
- [27] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, 2020.
- [28] Junyang Lin, Rui Men, An Yang, Chang Zhou, Ming Ding, Yichang Zhang, Peng Wang, Ang Wang, Le Jiang, Xianyan Jia, Jie Zhang, Jianwei Zhang, Xu Zou, Zhikang Li, Xiaodong Deng, Jie Liu, Jinbao Xue, Huiling Zhou, Jianxin Ma, Jin Yu, Yong Li, Wei Lin, Jingren Zhou, Jie Tang, and Hongxia Yang. M6: A chinese multimodal pretrainer, 2021.
- [29] Junyang Lin, An Yang, Jinze Bai, Chang Zhou, Le Jiang, Xianyan Jia, Ang Wang, Jie Zhang, Yong Li, Wei Lin, et al. M6-10t: A sharing-delinking paradigm for efficient multi-trillion parameter pretraining. *arXiv preprint arXiv:2110.03888*, 2021.

- [30] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. *arXiv preprint arXiv:2103.14030*, 2021.
- [31] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.
- [32] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.
- [33] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters. *arXiv preprint arXiv:2104.04473*, 2021.
- [34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.
- [35] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.
- [36] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.
- [37] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. *arXiv preprint arXiv:2104.07857*, 2021.
- [38] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.
- [39] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. *arXiv preprint arXiv:2101.06840*, 2021.
- [40] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.
- [41] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pages 10414–10423, 2018.
- [42] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR, 2018.
- [43] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- [44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.
- [45] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54, 2021.
- [46] Minjie Wang, Chien-Chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 26:1–26:17. ACM, 2019.
- [47] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous gpu clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, 2022.

- [48] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.
- [49] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 595–610. USENIX Association, 2018.
- [50] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, November 2020.
- [51] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. Gspmd: General and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.
- [52] Linting Xue, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, Aditya Siddhant, Aditya Barua, and Colin Raffel. mt5: A massively multilingual pre-trained text-to-text transformer. *arXiv preprint arXiv:2010.11934*, 2020.
- [53] An Yang, Junyang Lin, Rui Men, Chang Zhou, Le Jiang, Xianyan Jia, Ang Wang, Jie Zhang, Jiamang Wang, Yong Li, et al. Exploring sparse expert models and beyond. *arXiv preprint arXiv:2105.15082*, 2021.
- [54] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems*, 3, 2021.
- [55] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879, 2020.
- [56] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. Astitch: Enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2022.



Cachew: Machine Learning Input Data Processing as a Service

Dan Graur
ETH Zurich

Damien Aymon
ETH Zurich

Dan Kluser
ETH Zurich

Tanguy Albrici
ETH Zurich

Chandramohan A. Thekkath
Google

Ana Klimovic
ETH Zurich

Abstract

Processing input data plays a vital role in ML training, impacting accuracy, throughput, and cost. The input pipeline, which is responsible for feeding data-hungry GPUs/TPUs with training examples, is a common bottleneck. Alleviating data stalls is critical yet challenging for users. While today’s frameworks provide mechanisms to maximize input pipeline throughput (e.g., distributing data processing on remote CPU workers and/or reusing cached data transformations), leveraging these mechanisms to jointly optimize training time and cost is non-trivial. Users face two key challenges. First, ML schedulers focus on GPU/TPU resources, leaving users on their own to optimize multi-dimensional resource allocations for data processing. Second, input pipelines often consume excessive compute power to repeatedly transform the same data. Deciding which source or transformed data to cache is non-trivial: large datasets are expensive to store, the compute time saved by caching is not always the bottleneck for end-to-end training, and transformations may not be deterministic, hence reusing transformed data can impact accuracy.

We propose Cachew, a fully-managed service for ML data processing. Cachew dynamically scales distributed resources for data processing to avoid stalls in training jobs. The service also automatically applies caching when and where it is performance/cost-effective to reuse preprocessed data within and across jobs. Our key contributions are autoscaling and autocaching policies, which leverage domain-specific metrics collected at data workers and training clients (rather than generic resource utilization metrics) to minimize training time and cost. Compared to scaling workers with Kubernetes, Cachew’s policies reduce training time by up to $4.1\times$ and training cost by $1.1\times$ to $3.8\times$.

1 Introduction

Input data processing is an essential part of machine learning (ML) training. Transformations applied to input data before it is fed to a model for training – such as extracting features, sampling data from imbalanced classes, and randomly augmenting data – are key to achieving high accuracy [19,57,63]. Furthermore, the speed at which the input pipeline can ingest data from storage, apply transformations on-the-fly, and load

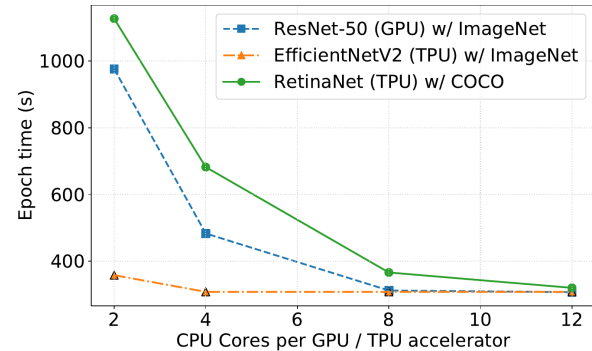


Figure 1: Training jobs benefit differently when given more CPU resources for input data processing per accelerator core.

transformed data to training nodes greatly impacts the time to accuracy and the overall cost of model training.

While GPUs and TPUs used for training computations continue to provide more FLOPS, CPUs – which are responsible for input data processing – are not keeping up. Hence, the input pipeline is a common bottleneck in ML training [38].

Removing bottlenecks in the input pipeline can improve end-to-end training time by over an order of magnitude and greatly reduce costs [47,48]. However, optimizing ML data processing is non-trivial. Users face several key challenges.

First, allocating the right amount of CPU, memory, and storage for input data processing, to optimize training time and cost, is difficult. Users should allocate *just enough* resources for the input pipeline to produce batches of data at the throughput that the model can ingest data, which depends on the model’s computational intensity and the hardware (# of GPUs) allocated for training. As shown in Figure 1, each model requires a different ratio of CPUs for data processing and GPUs or TPUs for training. Hence, although ML frameworks traditionally couple input data processing and training such that the two stages execute on the same node, it is becoming increasingly common to disaggregate data processing, with systems like `tf.data` service [25] and Meta’s Data PrePreprocessing (DPP) Service [69]. Disaggregation enables customizing resource allocations per job. However, today’s ML resource management systems focus on GPU allocations [28,44,65], leaving users on their own to decide input

pipeline resource allocations that maximize performance and minimize cost of training jobs. This is notoriously challenging in the multi-dimensional resource setting of ML training [64].

Another major challenge is the significant amount of compute and memory resources required for input data processing. For example, when training deep recommender models with petabytes of data, data ingestion can consume more power than model training itself [69]. A promising approach to reduce data processing compute requirements is to memoize the outputs of commonly executed data pipelines, since ML training often involves redundant data accesses and transformations, both within and across jobs [34, 47, 48, 66]. Within a job, it is common to iterate multiple times (i.e., epochs) over a dataset. Across jobs, ML engineers typically experiment with variations of models (e.g., hyperparameter tuning and model search) while re-executing the same data pipeline.

ML data processing frameworks provide mechanisms for reusing memoized data transformations, such as `tf.data`'s `cache` and `snapshot` operators [22, 61, 62]. However, determining which (transformed) datasets are optimal to cache in fast storage is non-trivial. Transformed datasets are costly to store and slow to read if they are significantly larger in volume than source data, which can occur after decompression and data augmentations. Figure 2 shows that reusing memoized results stored on local SSDs of a training node does not always improve epoch time, since local SSD bandwidth may saturate when reading the larger transformed dataset. Even caching source data on local SSDs does not always improve epoch time compared to reading from cloud data lakes (e.g., we use GCS [27]) since model training may be the bottleneck. If an input pipeline applies random transformations to data each epoch, the caching decision is further complicated as reusing the transformed dataset from a single epoch can negatively impact model training dynamics [16, 40].

In summary, ML data processing frameworks provide useful *mechanisms* to alleviate data bottlenecks, such as distributing data processing on remote CPU workers and reusing cached data transformations. However, today's systems lack *policies* that efficiently leverage these mechanisms to optimize the overall performance and cost of ML training.

We propose Cachew, a fully-managed service for ML input data processing. Inspired by the serverless computing paradigm, which relieves developers from the burden of managing virtual machines in the cloud [13, 55], Cachew relieves ML users from the burden of managing compute, memory, and storage infrastructure for ML input data processing. Cachew consists of a centralized dispatcher, distributed input data workers, and a disaggregated storage cluster that stores cached datasets. We build Cachew on top of the `tf.data` framework [48], extending its distributed service [25] to support multi-tenancy, autoscaling, and autocaching. Cachew is open source¹ and compatible with the existing `tf.data` API.

¹Cachew is available at: <https://github.com/eth-eas/cachew>

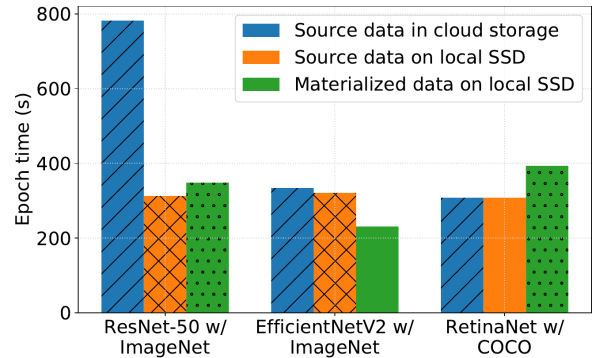


Figure 2: Caching source data or materializing data in local storage does not always improve training throughput.

Our key contributions are the design and implementation of autoscaling and autocaching policies for ML input data processing. We show that traditional resource autoscaling approaches (e.g., in Kubernetes [1]), which are based on CPU and memory utilization, are not sufficient to optimize training time and cost. Instead, we base our policies on application-specific metrics collected at input data workers and clients while a job is running. The Cachew dispatcher leverages the stateless nature of ML input data processing to adjust the number of workers per job during runtime. By monitoring batch time reported by clients, the dispatcher dynamically finds the minimum number of workers (i.e., minimum cost) that minimizes batch time (i.e., maximizes performance). By monitoring per-worker throughput and the volume of data produced by the input pipeline, the dispatcher also decides on whether reading data from Cachew's cluster cache is likely to improve performance compared to reading and transforming data from cloud data lakes on-the-fly. Cachew extends the `tf.data` API with an `autocache` operator, which allows users to specify up to which point in their input pipeline it is acceptable to cache and reuse data from a training dynamics perspective (e.g., before any random transformations).

We evaluate Cachew with microbenchmarks and three popular ML models and data pipelines from the TensorFlow Model Garden. We show that while the Kubernetes autoscaler under-provisions or over-provisions input data workers, Cachew is able to identify the optimal number of data workers to allocate for each job as well as the optimal caching strategy at each `autocache` operator location in the input pipeline to minimize training time and cost. We show that compared to scaling workers with Kubernetes, Cachew's policies reduce training cost by $1.1\times$ to $3.8\times$.

2 ML Input Data Processing

We summarize the key characteristics of ML input data pipelines (§ 2.1) and discuss why it is increasingly common to disaggregate data processing from model training (§ 2.2). § 2.3 provides an overview of existing mechanisms for fast, efficient input data pipeline execution.

2.1 ML Input Data Pipeline Characteristics

Reading input data from storage: The first step in ML input data processing is reading source data. Deep learning input pipelines typically read datasets that range from gigabytes to petabytes in size, stored in low-cost distributed storage systems, such as cloud data lakes [48, 68, 69]. To avoid I/O bottlenecks during training, it is common to cache input datasets in more expensive, higher bandwidth storage systems [39, 47].

Transforming data: Before raw input data can be consumed by a model, it must be preprocessed into elements that the model can learn from. Common transformations include decompressing data, parsing file formats, extracting features, and batching elements. It is also common to add randomness to input data (e.g., randomly sampling, augmenting, and shuffling elements) to improve model generalization [12]. Random data augmentations are critical for achieving state-of-the-art accuracy in image classification [14, 15, 18, 21, 58], object detection [19, 67], and speech recognition [49, 50].

Data transformations are generally executed on CPUs rather than specialized hardware to better support user-defined functions [48]. While some transformations can be applied in offline batch processing jobs [6], many transformations are applied *on-the-fly* during training for greater flexibility. For instance, it is common to experiment with feature extraction, tune the batch size for a given GPU configuration, or randomly augment data using different seeds across epochs.

While some transformations (e.g., decompression and augmentation) expand the volume of data read from storage, other transformations (e.g., filtering and sampling) decrease the volume of data fed to a model. A study of ML input pipelines at Google found that the ratio of data fed to a model versus the data read from storage varies widely across jobs. For 75% of jobs, data transformations reduced data volume [48].

Loading data to training nodes: The final step is to load data to GPUs/TPUs for model training. To avoid data stalls, the input pipeline must produce data at a throughput greater than or equal to the rate at which the model can consume data. The model's data ingestion rate depends on the algorithmic intensity of the training computations and the hardware FLOPS. Feeding data-hungry accelerators requires high software parallelism and pipelining for data processing [38, 48].

Re-executing input pipelines: In large-scale production and research deployments, ML input data pipelines are commonly re-executed. Within a job, each training epoch reads and transforms the same input dataset. Across jobs, common ML training workflows, such as neural architecture search and hyperparameter tuning, involve feeding the same pre-processed data to different variations of a model [34, 70]. At Google, the top 10% most commonly executed input pipelines accounted for 77% of all input pipeline executions and 72% of CPU cycles used for ML input data processing [48]. Others have also observed a sizeable opportunity to reuse data processing within and across ML jobs [47, 66].

2.2 Why disaggregate input data processing?

ML input data processing and training consist of fundamentally different types of computation (user-defined data transformations vs. gradient updates) that primarily use different resources (CPUs vs. GPUs/TPUs). Yet ML frameworks have typically coupled these two stages, such that they run on the same nodes. Tight coupling has two major drawbacks. First, CPU/memory-intensive input pipelines can easily saturate host resources and limit training throughput. Zhao et al. [69] showed that loading data over the network from distributed storage – even without performing data transformations on the CPU – consumes significant CPU cycles and memory bandwidth in production deployments, leaving scarce resources available for transforming data on training nodes. Second, the ratio of resources required for input data processing vs. model training varies across jobs (as seen in Figure 1, where we varied the CPU cores available using the Linux `taskset` command). However, cloud providers typically limit the CPU cores and memory capacity that can be provisioned per accelerator on a virtual machine [9, 26]. The fixed ratio of CPU cores and memory attached to accelerators often leads to imbalanced usage (i.e., either idle accelerators or idle CPUs).

To improve resource utilization and avoid input data stalls, it is increasingly common to disaggregate data processing from model training [25, 69]. Disaggregation is a well-known approach for improving resource allocation flexibility [23, 36]. This flexibility can save cost, since users can distribute input data processing across as many or as few CPU worker nodes as needed to avoid data stalls, without provisioning additional expensive GPUs/TPUs.

2.3 Existing Mechanisms

We describe key mechanisms in existing frameworks for efficient input data processing. Users can define and execute ML input data pipelines with a variety of data loading frameworks, such as `tf.data` [48], PyTorch DataLoader [17], NVIDIA DALI [29], and CoordDL [47]. We highlight the `tf.data` framework, as its combination of state-of-the-art mechanisms serve as a foundation for our work. `tf.data`'s programming model allows users to build input pipelines by composing and customizing operators. The framework runtime executes input pipelines as dataflow graphs, applying static and dynamic optimizations to improve performance.

Disaggregation: `tf.data` service supports executing input pipelines in a distributed manner [11]. The service consists of a centralized dispatcher and a number of remote input data workers. Clients (i.e., training nodes) register input pipelines defined in the `tf.data` API with the dispatcher, which shards data processing across all workers in the service. Clients fetch data directly from workers. The Data PreProcessing Service at Meta is another example of a framework that disaggregates input data processing [69]. In both frameworks, users are

responsible for managing the number of input data workers and deciding per-job resource allocations.

Dataset Caching: The `tf.data.snapshot` operator allows users to cache the output of their input pipeline to disk, and materialize the transformed data on a subsequent training run [22]. This trades off I/O capacity and bandwidth to free up CPU resources. Inserting the `snapshot` operator at the appropriate point in an input pipeline to optimize overall training time and cost remains the user’s responsibility. The current `snapshot` implementation does not coordinate between asynchronous reads and writes from multiple nodes, making it incompatible with `tf.data` service (hence, we implement our own `put` and `get` operators, described in §5.1).

CoorDL [47] and OneAccess [34] reuse input pipeline outputs when jobs are scheduled in a coordinated manner (e.g., hyperparameter tuning). However, these frameworks do not reuse data transformations across arbitrary ML jobs that can be submitted asynchronously to a service over time. Revamper [40] allows users to partially reuse the outputs of random transformations in input pipelines while minimizing the impact on training dynamics. Quiver [39] implements distributed caching for DNN training, but it is designed exclusively for managing source data rather than transformed datasets.

Autotuning: `tf.data`’s runtime and Plumber [38], a tool for diagnosing input data bottlenecks, can dynamically tune software parallelism and memory buffer sizes to maximize performance on a given training node [48]. However, this tuning does not scale resources beyond a single node.

GPU Offloading: NVIDIA DALI supports offloading certain input data processing, such as image data augmentations, to GPUs [29]. This is a viable option to alleviate CPU bottlenecks but may lead to GPU resource contention among input data transformation tasks and model training tasks. Users therefore need to decide which input data transformations should run on CPUs vs. GPUs.

3 ML Input Data Service Challenges

While many useful mechanisms for ML input data processing exist, it remains challenging for users to leverage these mechanisms to minimize end-to-end ML training time and cost. We focus on two key challenges: scaling resources for data processing (§ 3.1) and saving compute resources by selectively caching (transformed) datasets (§ 3.2).

3.1 Autoscaling Challenges

Selecting the right amount of compute, memory, and storage resources to provision for ML input data processing is critical yet challenging for ML users. Under-provisioning resources for data processing leads to data stalls, which leave expensive hardware accelerators idle, increasing end-to-end training time and cost. Over-provisioning resources for data

processing leads to extra costs without improving end-to-end performance. The optimal resource allocation for data processing depends on the compute intensity of data transformations in the input pipeline, the volume of data that must be read for each training batch, and the rate at which the pipeline must produce data to match model ingestion throughput.

Determining the right resource allocation for ML input data processing is non-trivial as each model and input data pipeline combination have unique requirements [69]. For example, in Figure 1, we vary the amount of CPU cores allocated for input data processing to show how many cores are needed to meet model training throughput requirements in various jobs. To process the COCO dataset [43] and train the RetinaNet [42] model, 4 CPU cores per TPU accelerator core are sufficient, whereas to process the ImageNet [21] dataset and train the EfficientNetv2 [59] model, the user should provision 12 CPU cores per TPU accelerator core to avoid input data stalls during training. We also observe that scaling memory capacity and bandwidth greatly impacts performance. Furthermore, we find that training throughput does not scale linearly with CPU and memory resources, making it difficult to model how resource allocation affects performance.

3.2 Autocaching Challenges

Deciding which input data transformations to materialize and reuse versus which transformations to execute online during training is complex. At Google, where `tf.data` is heavily used in research and production ML training jobs, only 19% of jobs use any kind of caching operator [48]. Meanwhile, the same study found that many jobs would benefit from caching.

Making caching decisions requires users to reason about the cost of storing preprocessed data to save CPU cycles. This trade-off depends on the compute intensity of the input pipeline, the size of the materialized dataset, and the relative cost of CPU and storage resources. Transformed datasets may be slow to read and costly to store if they are significantly larger in volume than source data, which can occur with decompression and data augmentations. Caching does not always improve epoch time, in particular if reading source data from cloud storage and transforming it on-the-fly is sufficiently fast to saturate model ingestion throughput. Another challenge is that caching and reusing the results of transformations which randomly permute data from one epoch will remove this randomness across epochs. Reusing this transformed data within a job can significantly impact training dynamics. Prior work has shown that reusing random augmentations across epochs in a job is plausible, but must be done sparingly to avoid degrading model accuracy [16, 40]. For example, Revamper proposes caching partially augmented data elements and mixing them with freshly-computed, fully-augmented elements [40].

An over-arching challenge is to jointly optimize autocaching and autoscaling. Regardless of whether input data

workers are reading and transforming source data on-the-fly from data lakes or reading transformed data from a cache, we need to determine the right number of input data workers and storage bandwidth to provision to maximize training throughput while keeping costs low.

4 Cachew Design

We introduce Cachew, a multi-tenant service for ML input data processing. To minimize end-to-end training time and cost, Cachew jointly optimizes: 1) elastic, distributed resource allocation for input data processing and 2) materialization of data processing computations within and across jobs. Cachew can be operated by an organization with multiple users that asynchronously submit ML training jobs or by a public cloud provider. With minimal extensions to the `tf.data` user API, Cachew transparently manages resource allocation for data processing, data caching, and network communication between Cachew clients and workers.

4.1 Service Architecture

Cachew consists of a centralized dispatcher, a dynamic number of input data workers, and a disaggregated storage cluster for data caching, as shown in Figure 3.

Users register training nodes (i.e., clients) of ML training jobs with the Cachew dispatcher. Clients provide a dataflow graph representation of their input pipeline and a path to the input data (§ 4.2). We assume input data resides in durable, low-cost cloud storage, i.e., data lakes [68] such as S3 [56].

Input data workers are stateless components responsible for producing batches of preprocessed data for clients. The dispatcher dynamically adjusts the number of input data workers for each job and divides each job’s input dataset (e.g., a list of filenames) into independent partitions, called *splits*. Workers pull new splits (e.g., indexes of the file list) from the dispatcher when they are done processing previous splits. Workers may read splits that correspond to source data which they must transform on-the-fly by executing the job’s input pipeline dataflow graph. Alternatively, splits may correspond to files that contain already transformed (or partially transformed) data in Cachew’s cache from previous executions of the input pipeline.

The dispatcher maintains metadata about input pipeline executions across jobs to make worker allocation and data caching decisions. The scaling and caching policies (described in § 4.3) and § 4.4) rely on the metrics listed in Table 1, which the dispatcher aggregates in its *metrics metadata store*, indexed by job ids, input pipeline hashes and job names. Since there may be multiple workers and clients per job, metrics are averaged across clients and workers of the same job. The dispatcher also tracks which source and transformed datasets are cached. The *cache metadata store* maintains the location of cached datasets in Cachew’s cluster cache and is indexed

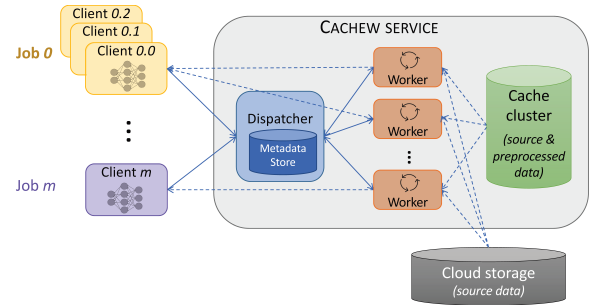


Figure 3: Cachew system architecture. Solid lines depict control logic and metadata communication. Dotted lines show the flow of training data. Communication occurs via RPCs.

```

1 dataset = tf.data.TFRecordDataset(["file1", ...])
2 dataset = dataset.map(parse).filter(filter_func)
3                 .autocache()
4                 .map(rand_augment)
5                 .shuffle().batch()
6 dataset = dataset.apply(distribute(dispatcherIP))
7 for element in dataset:
8     train_step(element)

```

Figure 4: User API to distribute `tf.data` input pipeline execution with Cachew. Users insert `autocache` to hint which data is acceptable to cache/memoize and reuse within a job.

by various hashes of the input pipeline dataflow graph, which we call fingerprints.

Clients fetch data from the workers that are assigned to them by the dispatcher. Clients and workers periodically send heartbeats to the dispatcher (by default every five seconds) to maintain membership in the service and provide metrics.

Cachew’s cache cluster consists of high-bandwidth NVMe SSD storage nodes, which are disaggregated from input data workers. Hence, Cachew can scale storage independently, based on data caching capacity and bandwidth requirements. In addition to caching transformed datasets of frequently executed input pipelines, Cachew can also cache source datasets, to avoid I/O bottlenecks from cloud data lakes during training.

4.2 Cachew API

Cachew leverages the existing `tf.data` API for defining ML input data pipelines [48]. Users define a pipeline by chaining dataflow operators that can be parameterized with user-defined functions (UDFs). Figure 4 shows an example `tf.data` pipeline that reads input data from files, applies `map` and `filter` operators with UDFs for parsing, filtering, and randomly augmenting data, then shuffles and batches data.

Applying `distribute` in line 6 serializes the dataflow graph and sends it to the service dispatcher to register the job. If there are multiple training nodes in a job (i.e., for dis-

Source	Name	Description
Client	batch_time	Time taken to get and process the last 100 batches
	result_queue_size	Avg. number of batches located in the prefetch buffer over the last 100 batches
Worker	active_time	Avg. time per element spent in computation in the subtree rooted in the node
	bytes_produced	Total number of bytes produced by the node so far
	num_elements	Total number of elements produced by the node so far

Table 1: The set of metrics that are submitted by the workers and clients to the dispatcher via heartbeats.

tributed ML training), each node registers as a separate client with the dispatcher and specifies an additional `job_name` parameter that is common across all clients of the same job. The communication between clients and workers is abstracted away from the python API. As shown in line 7, clients simply iterate over the `dataset` to access a sequence of elements, as if executing the input pipeline locally on the client.

Cachew introduces a new operator, `autocache`, which allows users to specify point(s) in an input pipeline where model training dynamics safely permit memoizing and reusing data within a job. To avoid any impact on training accuracy, users should apply `autocache` before any random data transformations in their pipeline (e.g., see line 3 of Figure 4). Users may apply `autocache` in multiple locations in a pipeline. Cachew will not always apply caching at an `autocache` operator; § 4.4 describes Cachew’s decision strategy. When `autocache` is placed after a read operator (e.g., line 1), Cachew decides whether caching source data in fast cluster storage improves performance compared to reading source data from a low-cost data lake. In §6, we evaluate Cachew with up to two `autocache` operators per pipeline.

4.3 Autoscaling Policy

We describe how Cachew leverages the per-job client metrics described in Table 1 to make worker scaling decisions.

Allocating workers for new jobs: The dispatcher starts by executing each new job with a single worker. After the client reports metrics from a configurable number of training batches, the dispatcher allocates a second worker for the job and monitors the change in `batch_time`. We observe that averaging metrics over 100 batches generally provides a satisfactory level of noise smoothing. If `batch_time` decreases by more than a threshold (which we empirically set to 3%) with the second worker, the dispatcher adds an additional worker. The dispatcher continues adding workers until a new worker improves `batch_time` by less than 3%, in which case the scaling decision converges. The epoch time (and hence batch time) plateaus when the workers can provide data at sufficient throughput to saturate the model ingestion rate (e.g., the red dotted line in Figure 6). The true plateau occurs when the addition of a new worker leads to 0% change in `batch_time`. However, we choose a slightly higher threshold due to the noisy nature of metrics gathered at runtime.

This makes Cachew’s autoscaling policy more stable. We observed that a 0% threshold would lead to unstable scaling decisions, as the slightest noise could trigger the addition of superfluous workers or the removal of essential workers from the cluster. In order to reduce noise, it is possible to gather the metrics over more batches, however this slows down the scaling process. §6.2 shows a sensitivity study.

Re-scaling over time: Since metrics can be noisy, Cachew periodically revisits the scaling decision each 10 new metrics received from the client (i.e., every 1000 batches). Cachew adds a worker if the current `batch_time` is significantly higher than the value recorded at scaling convergence. To detect if we are on the batch time plateau and can afford to scale down, our intuition is that although batch time will be the same, the result queue will build up if workers are able to provide data faster than the model can ingest it. Hence, Cachew removes a worker if the current `result_queue_size` is significantly (e.g., > 40%) higher than the size recorded at convergence. Cachew continues removing workers as long as the increase in `batch_time` is below the threshold. Clients temporarily pause metric collection (e.g. 150 batches) when they are notified that a worker has been added or removed for their job. When the dispatcher de-allocates a worker from a job, the worker processes any remaining splits, which clients consume before the worker is removed. Hence, the model sees all data in an epoch, regardless of scaling events.

We observed that in some cases, when the input dataset consists of few files (e.g., the COCO dataset), the scaling policy may not converge within an epoch because workers may prefetch all dataset splits, leaving no splits for newly added workers to process until the next epoch. Cachew detects these scenarios and inserts an artificial epoch, which allows workers to fetch data from the next epoch while other workers process data from the current epoch. We do not introduce artificial epochs if workers are writing to cache.

4.4 Autocaching Policy

When a new job is sent to Cachew, a hash of the entire input pipeline is generated, which is then used to check if the given pipeline has ever had some of its data cached. If yes, the dispatcher extracts a hash for each `autocache` op. This is done by traversing and hashing the input pipeline from its source nodes up until the `autocache` node. Cachew checks its

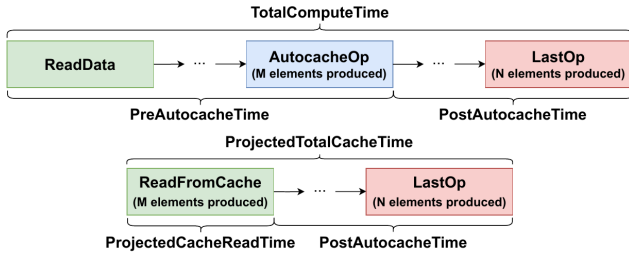


Figure 5: Cachew autocaching policy calculation.

cache store against these hashes to find potential hits. Cachew will choose to introduce caching at the `autocache` location with the highest throughput. It should be noted that in such cases, compute is not considered, as caching only occurs if it was deemed better in terms of throughput than compute. If the given pipeline has not got any of its data in cache, the job enters profiling mode. This stage is equivalent to full input pipeline computation, with the exception that scaling is blocked to not skew any metrics relevant to the autocaching decision. The relevant metrics are presented in the Worker section of Table 1. Once the input pipeline has produced a sufficient amount of batches (we observe 300 batches to be enough) Cachew’s dispatcher produces the autocaching decision.

To carry out the autocaching decision, for each `autocache` operator, the dispatcher estimates the time it takes the pipeline to produce N elements if caching were to be introduced at the `autocache` location. The dispatcher compares these N -element-times with the compute mode N -element-time (i.e., how long it takes to produce N elements if no caching is applied to the pipeline) and selects the option with the minimum time. If caching is chosen, Cachew introduces caching at the relevant `autocache op` location. Figure 5 shows the main values inferred by the dispatcher for the autocaching decision. It should be noted that due to operations such as `batch`, `filter`, or `repeat`, an `autocache op` sees M elements being produced, where $M \neq N$ can be true.

Let the `LastOp`’s `active_time` be `active_timeL`, an `autocache`’s `active_time` be `active_timeA` and the bytes per element at an `autocache op` be b_A . Cachew employs a throughput model of GlusterFS, formally defined as $g_{GFS} : \mathbb{R}_{>0} \rightarrow \mathbb{R}_{>0}$, which can infer the read time of b bytes from cache. Cachew initially computes the `TotalComputeTime` = $N \times \text{active_time}_L$. For each `autocache op`, Cachew computes the `PreAutocacheTime` = $M \times \text{active_time}_A$, and then obtains the `PostAutocacheTime` = `TotalComputeTime` - `PreAutocacheTime`. Next, Cachew computes `ProjectedCacheReadTime` = $M \times g_{GFS}(b_A)$. Finally, the `ProjectedTotalCacheTime` = `ProjectedCacheReadTime` + `PostAutocacheTime` is computed. The `ProjectedTotalCacheTime` of each `autocache op` and the initial `TotalComputeTime` are compared, and the

option with the lowest value is selected.

Once the decision is made, scaling is re-enabled. Scaling is triggered when the execution policy of the input pipeline changes (e.g. from putting to getting data from cache). When this happens, the worker count of a job is set to 1, and the autoscaling policy is applied by initially scaling up.

4.5 Multi-tenancy

Cachew supports multi-tenancy. Jobs submitted by different clients are accepted by Cachew, whose dispatcher applies the autoscale and autocache policies on each job. The jobs are self contained, and the autoscale and autocache decisions are independent of other jobs running in Cachew. To ensure such decisions are correct, metrics from different jobs with identical input pipelines are stored in the metadata store separately from one another using the job names. Moreover, to avoid performance interference, Cachew assigns each workers to at most one job at a time.

Our current prototype of Cachew assumes that tenants are mutually trusted and have permission to access each other’s data. Hence, Cachew shares cached datasets across jobs from different tenants. In §7, we discuss how the implementation can be extended to implement data access control.

4.6 Fault tolerance

Dispatcher: The dispatcher stores metadata in memory for fast look-ups. To avoid being a single point of failure, the dispatcher can journal its state to a durable directory, such that no state is lost when the dispatcher is restarted. Journaling is also supported in the vanilla `tf.data` service [25].

Workers: Our implementation builds on top of existing single-node `tf.data` checkpointing mechanisms, extending them to work in the distributed setting. Workers communicate with the dispatcher via heartbeats. If a parametrizable number of heartbeats are missed (we set this parameter to 2), the dispatcher considers the worker failed and initiates a failover. The dispatcher reassigns pending tasks to a free worker in the pool. The new worker recovers any progress the failed worker had made on the pending tasks by reading the latest checkpoint committed in remote storage. The new worker recomputes batches between the latest checkpoint and the time of failure, to make the new worker’s iterator state match the old worker’s iterator state at the time of failure. The client includes an incrementing index in its request which the new worker uses as time of failure to fast forward to. However, the new worker does not transmit the recomputed batches to the client since repeating data elements in training epochs can harm model accuracy. For instance, Mohan et al. observed a double-digit drop in Top-1 accuracy when training a ResNet18 for 70 epochs on ImageNet-1k without exactly-once semantics [46]. In contrast, Cachew guarantees that clients see each batch of input data exactly once during

training, even in the face of failures. Furthermore, our fault-tolerance mechanisms make it viable to run Cachew workers on transient cloud resources (i.e., spot VMs) to reduce the cost of using remote workers for data processing.

Storage nodes: Cachew’s distributed cache applies erasure coding with configurable redundancy. By default, we configure the storage layer to store data with sufficient redundancy to handle up to 25% of nodes in the storage cluster failing at a given time. If a storage node fails, the distributed file system uses parity blocks to recover data.

5 Implementation

We implement Cachew on top of the `tf.data` ML data processing framework [48], leveraging its familiar API and mechanisms for distributed data processing and dataflow graph rewriting. We also add a scalable cluster cache to the vanilla `tf.data` service and expose the `autocache` operator to users in the API. We extend the `tf.data` service dispatcher with metadata stores that manage client/worker metrics and locations of cached datasets to implement our scaling and caching policies. Our implementation consists of approximately 9000 lines of C++ code and 500 lines of Python on top of the open-source `tf.data` code base. Cachew is open-source.

We run the Cachew dispatcher and workers inside Docker containers and use Kubernetes to elastically scale the deployment. All communication between clients, workers, and the dispatcher is done over gRPC [24]. We use GlusterFS [53], deployed on high-bandwidth NVMe SSD storage nodes, as our distributed caching storage system. GlusterFS is highly scalable, offers sufficient throughput to saturate NVMe SSD bandwidth, and its consistent hashing data distribution policy supports dynamically adding and removing storage nodes. We configure GlusterFS with distributed dispersed volumes, which use erasure coding for fault-tolerance [54].

5.1 Autocache Mechanisms

Graph rewrites: When a user registers a new job, Cachew’s dispatcher inspects the input pipeline’s dataflow graph. Whenever the pipeline contains an `autocache` op, Cachew generates two versions of the pipeline, in which the `autocache` is replaced by a `put` op and a `get` op, respectively. The dispatcher transparently replaces `autocache` with graph rewrites using the TensorFlow Grappler [60] optimization framework. The dispatcher sends workers the correct version of the input pipeline graph, which depends on the execution mode selected by Cachew’s autocaching policy for the job.

The put and get ops: Cachew introduces `put` and `get` ops which store and retrieve data to and from the cache. These ops build on underlying mechanisms in the `tf.data` `snapshot` op implementation, but are designed for multi-worker scenarios, where several different worker nodes can concurrently write and/or read to the same cache location

without conflicts. To support asynchronous behaviour, the `put` and `get` op implementations both leverage queues and multi-threading. For the `put` op, data to be written to cache is placed in a queue. Multiple threads greedily dequeue elements and write them to cache. Each thread writes in its own file, which is closed when the size exceeds 250MiB and a new file is opened. The `get` op functions in the opposite manner, where threads read from cache and place elements in the queue. When downstream operations require data from the `get` op, they dequeue elements. Each thread requests the file paths to read from the dispatcher. Operations before the `get` op are not executed since the output is read from cache.

Since the ops are asynchronous, reads and writes will be performed out of order. Hence, cache reads and writes behave as a sliding-window shuffle of size $w_s = s \times n$, where w_s is the window size, s is the number of elements per cache file and n is the number of readers or writers.

Dealing with limited cache capacity: Our current prototype assumes that Cachew’s cache capacity is unbounded (i.e., there is always space in the cache for a `put` op to succeed). To operate Cachew with a limited cache size, we plan to extend the dispatcher implementation to periodically evict cached datasets that provide the least performance improvement across jobs, as in prior caching systems like Nectar [30].

6 Evaluation

6.1 Methodology

Workloads: We evaluate Cachew with three popular ML models and their corresponding input data pipelines in TensorFlow Model Garden [3]. ResNet-50 [31] is an image classification model whose input pipeline consists of parsing raw TFRecord files, converting images to float16 format, and applying a random crop and horizontal flip [4]. We use ImageNet [21] and observe that the data transformations increase the source dataset by 2.6×. RetinaNet [42] is an object detection model whose input pipeline consists of parsing TFRecords, converting images to float16, applying a random horizontal flip, and a series of computationally-intensive operations that create candidate anchors at five different scale levels [5]. We train RetinaNet on the COCO [43] dataset and the data transformations increase the data volume by 32.6×. Finally, SimCLRv2 [15] is a semi-supervised learning framework used for visual representation learning model. Given a randomly sampled mini-batch of images, each image is augmented twice with a random crop, color distortion, and Gaussian blur, creating two views of the same example. The model learns representations by maximizing agreement between differently augmented views of the same data example [2]. We use SimCLR for semi-supervised image classification on ImageNet. The data transformations increase the data volume by 10.7×.

Baselines: We compare Cachew’s resource scaling policy with the Kubernetes Horizontal Pod Autoscaler (HPA)

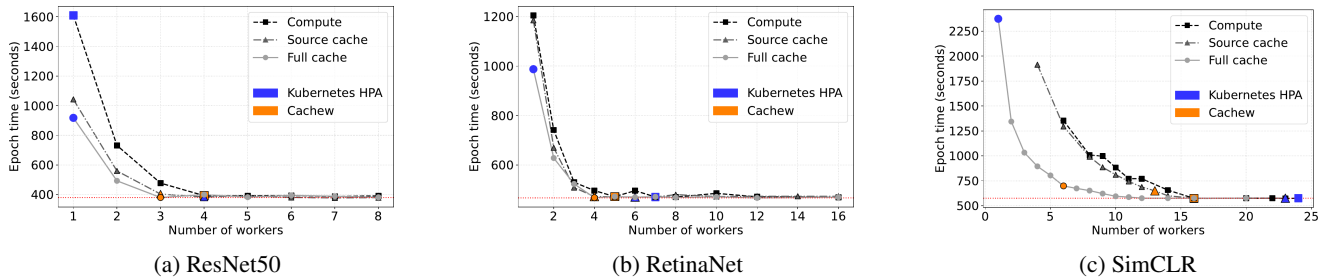


Figure 6: Scaling policy. Cachew selects the right number of workers to minimize epoch time and cost (orange markers). Kubernetes Horizontal Pod Autoscaler does not select the optimal number of workers (blue markers), since it only scales based on CPU usage and does not account for other potential input pipeline bottlenecks, e.g., memory and I/O bandwidth.

policy, which scales input data workers in the service based on a 80% CPU resource utilization target per node [1]. We measure the best-case epoch time for each model (i.e., model ingestion rate) by running an infinitely fast input pipeline that feeds synthetic data. Finally, we report the overhead of running `tf.data` pipelines on remote workers versus on training nodes.

Execution modes: We consider two different placements of the `autocache` operator in our input pipelines. We assume the user inserts `autocache` near the beginning of the input pipeline, immediately after a data read operator and before any data transformation operators. We call this placement *source cache mode* since it causes data workers to read source data from Cachew’s cluster cache if the dispatcher decides to apply caching at this point in the pipeline. We also assume the user inserts `autocache` at the end of the input pipeline, after all data transformations. We refer to this placement as *full cache mode*, since input data workers will read fully transformed data from the Cachew cluster cache if the dispatcher decides to apply caching at this point in the pipeline. We compare the performance of the source cache and full cache execution modes with a *compute mode*, in which no data is reused in the input pipeline, i.e., Cachew reads source data from cloud storage and transforms data on-the-fly.

Metrics: We measure epoch time, i.e. the time it takes to train the model on a complete iteration of the dataset, for each model while varying the number of input data workers and execution modes. We also report total training time and cost for the compute and source cache execution modes. For cost, we consider the Cachew input data worker node costs, storage resources, and the cost of training nodes used for the duration of the job. We assume the cost of the dispatcher is amortized across multiple users and jobs. Note that in our workloads, we observed full caching decreases training accuracy due to the presence of random transformations in the input pipelines. In our evaluation, we focus on demonstrating that Cachew can select the right execution mode to maximize throughput, assuming the user has placed the `autocache` operator in a location that is acceptable for their use-case. Prior studies

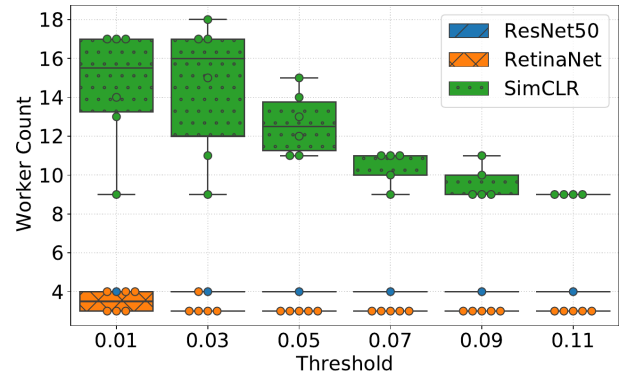


Figure 7: Cachew’s first scaling decisions in compute mode relative to the value of the improvement threshold.

have explored the impact on training dynamics when reusing randomly transformed data across epochs [16, 40].

Cluster hardware setup: We run our experiments on Google Cloud. The dispatcher runs on a n2-standard-16 VM. We train ResNet-50 on n1-standard-32 VMs with four Tesla V100 GPUs. We train SimCLRv2 and RetinaNet models on v3-8 TPU VMs since the reference implementations are designed for training on TPUs. We use n2-standard-2 VMs with two 375GB NVMe SSDs for the GlusterFS storage cluster. The network bandwidth between the storage cluster, workers, and clients is at least 16 Gb/s.

6.2 Cachew Autoscaling

We sweep the number of input data workers for the compute, source cache, and full cache execution modes for each model. Figure 6 plots epoch time as a function of the number of data workers and shows the number of workers selected by Cachew’s scaling policy in orange and Kubernetes’s scaling policy in blue markers. The dotted red line shows the minimum epoch time achievable with an infinitely fast input pipeline. In all execution modes, Cachew finds the minimum (or near minimum) number of data workers to avoid data stalls

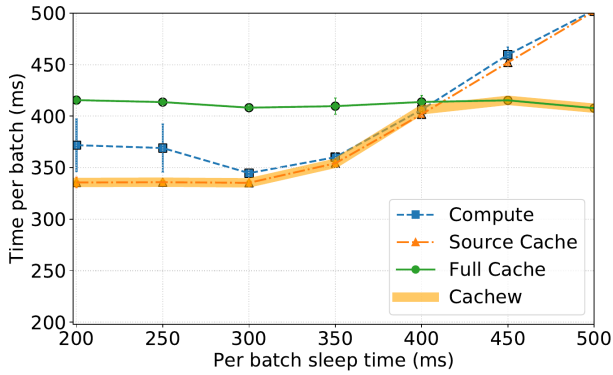


Figure 8: Cachew’s autocaching policy selects the execution mode that minimizes batch time.

in model training. In contrast, the Kubernetes Horizontal Pod Autoscaler noticeably under or over-provisions data workers. Kubernetes HPA performs poorly as it does not detect bottlenecks besides CPU and memory capacity, such as memory bandwidth bottlenecks and I/O bottlenecks that can limit input pipeline throughput. For SimCLRv2, the input pipeline is highly compute intensive, hence Kubernetes scales up to 24 data workers to maintain per-node CPU utilization at the 80% target. In contrast, Cachew’s scaling policy checks the relative improvement in batch time and determines there are diminishing returns to scaling beyond 16 workers in compute mode and 13 workers in source cache mode.

Figure 7 shows Cachew’s scaling decision sensitivity to the `batch_time` improvement threshold value, which we sweep from 1% to 11%. We include decisions that are suboptimal, which Cachew may later correct in the training process. Both ResNet50 and RetinaNet are robust to this threshold, as the addition of a worker yields clear epoch time benefits (see Figures 6a and 6b), thus the value of the threshold can be quite large. For SimCLR, the addition of a new worker does not always yield significant benefits to the epoch time (see Figure 6c). Consequently, for such a model, lower thresholds are more suitable. The downside of a lower threshold is that the autoscaling policy becomes more susceptible to noise in the metrics. This is visible in Figure 7, as the variance of the decision increases, and outlier decisions become more common. Cachew triggers rescaling in such cases, and eventually converges to the right decision. Gathering metrics over multiple batches can help alleviate noise.

6.3 Cachew Autocaching

To evaluate Cachew’s caching policy, we run the service with an input pipeline in which we carefully control and simulate compute intensity. The input pipeline consists of reading input data from storage (56GB), increasing the size of data elements by 2.5×, and sleeping for a controlled duration of time to simulate applying a time-consuming data processing

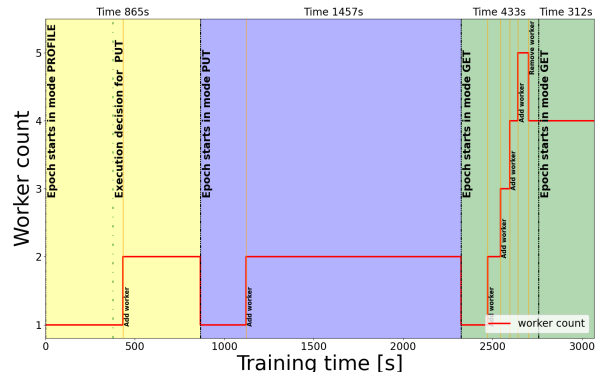


Figure 9: RetinaNet training timeline for the first 4 epochs. Cachew picks the right caching mode and number of workers.

operation. Figure 8 plots the time it takes to process a batch of elements as a function of the injected processing time, for the three different execution modes we consider. The experiment is designed such that the compute, source cache, and full cache execution modes are each optimal in a particular regime and verify that Cachew makes the optimal choice.

At low data processing intensity, reading data from the cloud data lake (GCS) is an I/O bottleneck and Cachew recognizes the service should store and read source data from the cluster cache. When sleep exceeds 350ms, Cachew recognizes that data processing intensity is high enough that GCS I/O is no longer the bottleneck. Reading and transforming data on-the-fly from GCS (i.e., compute mode) becomes optimal until 400ms. When source cache mode reaches a similar throughput as compute mode (as in the 350-400ms regime), Cachew prefers compute mode as it saves storage costs. When sleep time exceeds 400ms, storing and reading the transformed dataset from cache minimizes batch time compared to other modes. Cachew chooses full caching in this regime.

6.4 Autocaching & Autoscaling over Time

We demonstrate how Cachew jointly optimizes elastic scaling and caching to maximize epoch time and cost. Figure 9 plots the scaling and caching decisions that Cachew makes over time. As an example, we show the first four epochs of RetinaNet training, where we place two `autocache` ops: one after the reading ops, and one at the end of the input pipeline. The red curve shows the number of workers used for the job and orange vertical lines show when Cachew makes a scaling decision. Cachew starts by processing the pipeline in compute mode with a single worker (highlighted in yellow), first blocking scaling and profiling the worker. During the single-worker profiling phase, Cachew collects metrics to make a caching policy decision and decides on caching at the end of the input pipeline. This decision takes place at the time marked by the green vertical dashed line. The caching decision is applied at the end of the first epoch, when we enter the `put` epoch

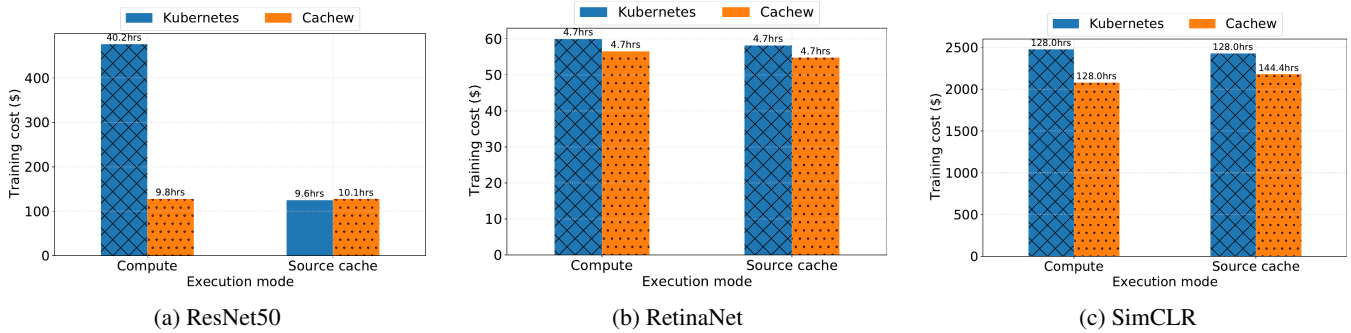


Figure 10: Total training cost (and training time) for Cachew vs. Kubernetes HPA worker scaling policy decisions.

(highlighted in blue) which writes the dataset to the cluster cache and takes longer as a result. Due to prefetching, splits are exhausted in the first two epochs before the autoscale completes. The other two epochs presented are in cache *get* mode (highlighted in green). In the third epoch, Cachew scales the number of workers up to 5, where no more improvements are observed. At this point Cachew removes the superfluous worker and converges to 4 workers. Cachew will continue to run the pipeline in *get* mode with 4 workers for the remaining epochs, and only adjust this number if input pipeline characteristics evolve, requiring less or more workers.

6.5 End-to-end performance and cost

6.5.1 Time to Accuracy and Training Cost

Figure 10 shows the total training cost (bars on y-axis) and total training time (annotations) for the input data worker configurations that Kubernetes and Cachew select in the compute and source cache execution modes. As we saw in Figure 6, Kubernetes under-provisions input data workers for ResNet50 compute mode, while Cachew picks the optimal worker count, allowing it to reduce training time by $4.1\times$ and cost by $3.8\times$. For SimCLR, Kubernetes over-provisions workers, slightly reducing training time but Cachew still saves overall cost by 12-20% by optimizing the number of workers based on its relative improvement threshold in the scaling policy.

6.5.2 Service Overhead

We compare the performance and resource overhead of running *tf.data* pipelines with the service versus locally on training nodes. The service requires additional CPU resources to achieve the same input pipeline throughput, due to the extra network hop (i.e., gRPCs) between data workers and training nodes. Since local *tf.data* workers fetch source data over the network and transform data on the training node, whereas Cachew clients fetch transformed data over the network, the service overhead also depends on the difference between the source and transformed dataset sizes. In our experiments, we find that the service should provision 30% to 50% more CPU

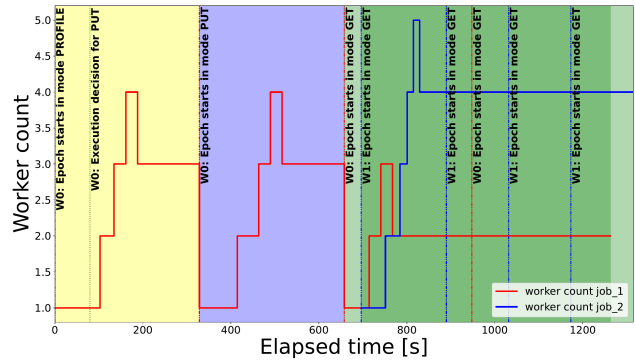


Figure 11: Multi-tenancy: Cachew detects cache hits across jobs while scaling workers for each job separately.

cores compared to the CPU cores we observe being heavily utilized when running the same input pipeline locally on training nodes. Note that this overhead is not an artifact of Cachew’s scaling or caching policies, but rather a measurement of the overhead Cachew inherits from the distributed data processing mechanisms in *tf.data* service [25]. Data marshaling is known to consume significant cycles in datacenters [35]. Networking overheads can be reduced (e.g., by using RDMA), though this is not the focus of our work. The cost of using extra CPUs for data processing is justified as it allows keeping expensive GPUs/TPUs highly utilized and reducing end-to-end training time (see Figure 10).

6.6 Multi-tenancy

We show that Cachew can optimize input data processing across jobs. We run two jobs with the ResNet input data pipeline but different model ingestion rates (simulated with different sleep times in the dataset iteration loop on the clients). The second job has double the ingestion rate of the first. We again place *autocache* ops after reading the data and at the end of the input pipeline. Figure 11 shows how the batch processing time varies over time for each job as Cachew makes its scaling and execution mode decisions. The red

curve shows the first job’s number of workers over time. The epoch boundaries are highlighted by the red dotted vertical lines. This job progresses through a compute epoch (yellow highlight) with profiling, a put epoch (blue highlight), where it caches the data of the `autocache` at the end of the input pipeline and a get stage (green highlight), which consists of two get epochs. The job ultimately converges to two workers.

We show that the dispatcher detects a cache hit after the first job has written to cache and selects the cache `get` execution mode for the second job in all of its four epochs. The epochs are separated by blue dotted vertical lines. The blue curve shows the number of workers for the second job over time. As this job has double the ingestion rate requirements of the first job, it converges to four workers.

7 Discussion

Data access control: Our current prototype of Cachew assumes that clients have permission to access each other’s data (e.g., tenants are all members of the same organization). The implementation can be extended with Access Control Lists (ACLs) to prevent unauthorized access to the data. Workers can check/set permissions before reading/writing datasets. If a job attempts to read from a dataset for which the client does not have read permission, the worker’s request will fail and the client will get a permission error.

Model training dynamics: Though reusing the output of random data augmentations can negatively impact model accuracy, prior works have found that negative impacts can be mitigated by tuning the extent to which caching is applied in an input pipeline. Choi et al. showed that reusing data after random transformations has a small negative impact on the final accuracy trained models, reaching highly competitive out-of-sample error rates with fewer non-cached data instances than a model with no echoing [16]. Revamper [40] demonstrated that partially caching random transformations, while leaving some to be applied after reading from cache, has negligible accuracy penalties, as long as the downstream random transformations provide sufficient sample diversity. Hence, a good rule of thumb is to cache expensive random transformations, while applying highly diverse and inexpensive random transformations after the cache. For instance, in tasks such as image classification, inexpensive random transformations, such as random crop and flip, generally provide sufficient sample diversity [40].

Leveraging local resources on training nodes: Although our autoscaling policy has focused on leveraging remote CPU workers for data processing, `tf.data` service also supports running data processing on local workers, which execute on client training nodes. Adapting Cachew’s autoscaling policy to leverage a mix of local and remote workers would ensure that the extra cost of remote workers is only incurred if the CPU/memory resources for data processing on client training nodes are not sufficient to avoid data stalls.

8 Related Work

§ 2.3 discussed existing mechanisms in input data frameworks, which serve as the foundation for Cachew’s autoscaling and autocaching policies. We discuss other related work below.

Automated resource provisioning: Cluster management systems aim to automate resource allocation decisions, which are notoriously difficult for users [8, 20, 37, 45]. Generic cluster managers treat workloads as black boxes, making them unsuitable for jointly optimizing data caching and resource scaling decisions. Cachew is able to jointly optimize caching and scaling by using metrics that are tailored to the ML input data processing domain. Several resource management systems have been developed specifically for deep learning jobs [28, 44, 65]. However, they assume GPUs are the dominant resource for ML training, whereas recent work has shown that allocating resources for input data processing is equally important yet not well addressed by existing systems [64].

Caching vs. recomputing intermediate data: Caching data and memoizing data transformations is a common technique [10, 41, 52]. Trade-offs of caching vs. recomputing data arise in various contexts [7, 32, 33, 51]. We draw particular inspiration from Nectar [30], a datacenter-scale caching system that treats computations and their intermediate results as interchangeable. We address ML-specific challenges, where estimating the benefit of caching on training time is non-trivial since the model may be the bottleneck or the (transformed) dataset may be prohibitively large to cache. We also jointly optimize caching and resource scaling.

9 Conclusion

We proposed Cachew, a system architecture, to enable input data processing *as a service* for machine learning. To avoid input data stalls while minimizing cost, Cachew dynamically scales distributed data processing resources to match the rate at which each job’s training nodes can ingest data, while avoiding over-provisioning. Cachew leverages its centralized view of data processing pipelines across mutually trusted jobs to reduce the overall compute power required for data processing, by transparently reusing (transformed) datasets within and across jobs when performance and cost efficient.

Acknowledgements

We thank our anonymous reviewers and shepherd for their valuable comments. We acknowledge Jiří Šimša, Andrew Audibert, Gustavo Alonso, Petros Maniatis, Paul Barham, and Julia Bazińska for discussions that helped strengthen this work. We also thank Oto Mraz for his help with artifact evaluation. We are grateful for access to the Google TPU Research Cloud and generous support from a Google Research Award.

References

- [1] Kubernetes Horizontal Pod Autoscaler. <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>, 2021.
- [2] SimCLR reference code. <https://github.com/google-research/simclr>, 2021.
- [3] TensorFlow Model Garden. <https://github.com/tensorflow/models>, 2021.
- [4] TF Model Garden: ResNet50 reference code. https://github.com/tensorflow/models/tree/master/official/vision/image_classification/resnet, 2021.
- [5] TF Model Garden: RetinaNet reference code. <https://github.com/tensorflow/models/tree/master/official/legacy/detection>, 2021.
- [6] Data preprocessing for machine learning: options and recommendations. <https://cloud.google.com/architecture/data-preprocessing-for-ml-with-tf-transform-pt1>, 2022.
- [7] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Automated selection of materialized views and indexes in sql databases. In *In Proceedings of VLDB '00*, pages 496–505, 2000.
- [8] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, 2017.
- [9] Amazon Web Services. AWS EC2 Instance Types. <https://aws.amazon.com/ec2/instance-types/>, 2021.
- [10] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruva Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proc. of Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 267–280, 2012.
- [11] Andrew Audibert and Rohan Jain. tf.data Service RFC. <https://github.com/tensorflow/community/blob/master/rfcs/20200113-tf-data-service.md>, 2019.
- [12] Leon Bottou. Curiously Fast Convergence of some Stochastic Gradient Descent Algorithms. In *Proceedings of the Symposium on Learning and Data Science*, 2009.
- [13] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Commun. ACM*, 62(12):44–54, 2019.
- [14] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In Hal Daumé III and Aarti Singh, editors, *Proc. of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 1597–1607. PMLR, 13–18 Jul 2020.
- [15] Ting Chen, Simon Kornblith, Kevin Swersky, Mohammad Norouzi, and Geoffrey E Hinton. Big self-supervised models are strong semi-supervised learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 22243–22255. Curran Associates, Inc., 2020.
- [16] Dami Choi, Alexandre Passos, Christopher J. Shallue, and George E. Dahl. Faster Neural Network Training with Data Echoing, 2019.
- [17] Torch Contributors. PyTorch Docs: torch.utils.data. <https://pytorch.org/docs/stable/data.html>, 2021.
- [18] Ekin D. Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. Autoaugment: Learning augmentation strategies from data. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 113–123, 2019.
- [19] Ekin Dogus Cubuk, Barret Zoph, Jon Shlens, and Quoc Le. Randaugment: Practical automated data augmentation with a reduced search space. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, pages 18613–18624, 2020.
- [20] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14*, pages 127–144, 2014.
- [21] Jia Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Proceedings of CVPR*, 2009.
- [22] Frank Chen and Rohan Jain. tf.data Snapshot RFC. <https://github.com/tensorflow/community/blob/master/rfcs/20200107-tf-data-snapshot.md>, 2020.

- [23] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 249–264, November 2016.
- [24] Google. gRPC: a high performance, open source universal RPC framework. <https://grpc.io/>, 2021.
- [25] Google. Module: tf.data.experimental.service. https://www.tensorflow.org/api_docs/python/tf/data/experimental/service, 2021.
- [26] Google Cloud. Cloud TPU pricing. <https://cloud.google.com/tpu/pricing>, 2021.
- [27] Google Cloud. Google Cloud Storage. <https://cloud.google.com/storage>, 2021.
- [28] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, February 2019.
- [29] Joaquin Anton Guirao, Krzysztof Łęcki, Janusz Lisiecki, Serge Panev, Michał Szolucha, Albert Wolant, and Michał Zientkiewicz. Fast AI Data Preprocessing with NVIDIA DALI. <https://devblogs.nvidia.com/fast-ai-data-preprocessing-with-nvidia-dali>, 2019.
- [30] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proc. of Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.
- [31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of CVPR*, pages 770–778. IEEE Computer Society, 2016.
- [32] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *Proc. of Programming Language Design and Implementation (PLDI’00)*, page 311–320, 2000.
- [33] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. Selecting subexpressions to materialize at datacenter scale. *Proc. VLDB Endow.*, 11(7):800–812, March 2018.
- [34] Aarati Kakaraparthi, Abhay Venkatesh, Amar Phanshaye, and Shivaram Venkataraman. The case for unifying data loading in machine learning clusters. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.
- [35] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proc. of the 42nd Annual International Symposium on Computer Architecture, ISCA ’15*, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.
- [36] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proc. of European Conference on Computer Systems, EuroSys ’16*, pages 29:1–29:15, 2016.
- [37] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 759–773, Boston, MA, July 2018. USENIX Association.
- [38] Michael Kuchnik, Ana Klimovic, Jiri Simsa, George Amvrosiadis, and Virginia Smith. Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines. *CoRR*, abs/2111.04131, 2021.
- [39] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 283–296, Santa Clara, CA, February 2020. USENIX Association.
- [40] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyunggeun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. Refurbish your training data: Reusing partially augmented samples for faster deep neural network training. In *USENIX Annual Technical Conference (ATC’21)*, pages 537–550, 2021.
- [41] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing, SOCC ’14*, page 1–15, 2014.
- [42] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *IEEE International Conference on Computer Vision (ICCV)*, pages 2999–3007, 2017.
- [43] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft coco: Common objects in context. In *Proceedings of ECCV*, 2014.

- [44] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, February 2020.
- [45] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud. In *USENIX Annual Technical Conference (USENIX ATC 20)*, pages 189–203, 2020.
- [46] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, Fine-Grained DNN checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216. USENIX Association, February 2021.
- [47] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. In *VLDB 2021*, January 2021.
- [48] Derek G. Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. In *VLDB 2021*, volume 14, 2021.
- [49] Daniel S. Park and William Chan. SpecAugment: A New Data Augmentation Method for Automatic Speech Recognition. <https://ai.googleblog.com/2019/04/specaugment-new-data-augmentation.html>, 2019.
- [50] Daniel S. Park, William Chan, Yu Zhang, Chung-Cheng Chiu, Barret Zoph, Ekin D. Cubuk, and Quoc V. Le. SpecAugment: A simple data augmentation method for automatic speech recognition. *Interspeech 2019*, Sep 2019.
- [51] Lana Ramjit, Matteo Interlandi, Eugene Wu, and Ravi Netravali. Acorn: Aggressive result caching in distributed data processing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC’19, page 206–219, 2019.
- [52] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proc. of Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 401–417, 2016.
- [53] Red Hat. Gluster: scalable data filesystem. <https://www.gluster.org/>, 2021.
- [54] Red Hat. Setting up GlusterFS Volumes. <https://docs.gluster.org/en/v3/Administrator%20Guide/Setting%20Up%20Volumes/>, 2021.
- [55] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM*, 64(5):76–84, April 2021.
- [56] Amazon Web Services. Amazon Simple Storage Service. <https://aws.amazon.com/s3>, 2021.
- [57] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. *J. Big Data*, 6:60, 2019.
- [58] Patrice Y. Simard, Dave Steinkraus, and John C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of ICDAR*, page 958. IEEE Computer Society, 2003.
- [59] Mingxing Tan and Quoc Le. Efficientnetv2: Smaller models and faster training. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139, pages 10096–10106. PMLR, 2021.
- [60] TensorFlow. TensorFlow Graph Optimizations. <https://research.google/pubs/pub48051.pdf>, 2019.
- [61] Tensorflow. Better performance with the tf.data API. https://www.tensorflow.org/guide/data_performance#キャッシング, 2021.
- [62] Tensorflow. tf.data.experimental.snapshot. https://www.tensorflow.org/api_docs/python/tf/data/experimental/snapshot, 2021.
- [63] Jason Van Hulse, Taghi M. Khoshgoftaar, and Amri Napolitano. Experimental perspectives on learning from imbalanced data. In *Proceedings of the 24th International Conference on Machine Learning*, ICML ’07, page 935–942, 2007.
- [64] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Liping Zhang Yong Li, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, April 2022.
- [65] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, October 2018.

- [66] Doris Xin, Litian Ma, Jialin Liu, Stephen Macke, Shuchen Song, and Aditya Parameswaran. Helix: Accelerating human-in-the-loop machine learning. *Proc. VLDB Endow.*, 11(12):1958–1961, August 2018.
- [67] Sangdoon Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *International Conference on Computer Vision (ICCV)*, 2019.
- [68] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021*, 2021.
- [69] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding and co-designing the data ingestion pipeline for industry-scale recsys training. *CoRR*, abs/2108.09373, 2021.
- [70] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *Proceedings of ICLR*, 2017.

A Artifact Evaluation README

A.1 Abstract

The artifact consists of the source code of Cachew², the Cachew client binaries³, as well as scripts for building wheel files and Docker images. We also provide reference scripts for deploying GCE VMs for evaluation and for running the some representative experiments⁴. Do note that these scripts might not work as they depend on resources that might not be public. In these cases, experiment VMs will have to be manually set up.

The evaluation focuses on reproducing key experiments and their respective results which demonstrate how the main contributions of Cachew work:

- **Autoscaling** (*Figure 6a, compute curve*): show how input pipeline resources affect training time, how Cachew's autoscaling policy finds the right number of workers automatically, and how the Kubernetes Horizontal Pod Autoscaler fails to find the right scale.
- **Autocaching** (*Figure 8*): Show how Cachew's autocaching policy behaves under various execution scenarios, and how the most efficient execution mode is selected by Cachew.
- **Multi-tenancy with autoscaling and autocaching** (*Figure 11*): Show how Cachew behaves in multi-worker scenarios, selecting the most efficient execution mode, as well as the right scale for each job. Furthermore this experiment should also show how caching can be used in cross-job settings.

A.2 Artifact check-list

The artifact with the components listed below is available at: https://github.com/eth-easl/cachew_experiments.

- System to deploy: Cachew service (dispatcher, input data workers, remote cache cluster)
- Algorithms to evaluate: Cachew's autoscaling and autocaching policies
- Workloads to run:
 - *Figure 6a*: ResNet50 model and its open-source canonical input pipeline.
 - *Figure 8*: Synthetic input pipeline
 - *Figure 11*: Canonical ResNet50 input pipeline
- Binary: Cachew Docker image for workers and dispatcher, Cachew wheel file for client, GCE VM image.

²<https://github.com/eth-easl/cachew>

³gs://cachew-builds/tensorflow-2.8.0-cp39-cp39-linux_x86_64.whl

⁴https://github.com/eth-easl/cachew_experiments

- Model: ResNet50 and its canonical input pipeline
- Data Sets: ImageNet 2012 (stored in GCS bucket)
- Output: CSV files with metrics, text-based logs, and plots to compare with figures in the paper.
- Experiments: Experiments and deployment are fully scripted. See §A.4.
- Publicly available?: yes
- Code licenses: Apache 2.0
- Data licenses: ImageNet 3-Clause License
- Archived (provide DOI)? 10.5281/zenodo.6543943

A.3 Prerequisites

Hardware dependencies: The experiments require a cluster of x86 CPU servers with hardware virtualization support, with 4 Nvidia V100 GPUs on one of the servers. We recommend (and our scripts assume that you are) conducting experiments on Google Cloud. Some of the VM deployment scripts might not work out of the box as they can require access to resources which are no longer private. In this case, the scripts will either have to be modified or the deployment will have to be done manually.

Software dependencies: Our scripts make extensive use of the gcloud CLI tool. As a consequence, this tool is a prerequisite for setting up VMs and running experiments. Please follow [this tutorial](#) to install it. We additionally make use of the gsutil tool. To install it, please follow [this tutorial](#). We also suggest to use Python 3.9 with [PyEnv](#) as a means to install and manage multiple python versions and virtual environments. The [software requirements](#) for the Google Cloud service deployment are installed on the VM images we provide.

Estimated time and cost: The estimated time needed to prepare the workflow is 30 minutes. The estimated execution time of experiments is approx. 15 hours in total. We provide an estimated breakdown of the time and cost for each experiment⁵.

A.4 Instructions

Detailed instructions are provided in the [artifact repository README](#). The evaluator will need to `git clone https://github.com/eth-easl/cachew_experiments.git` locally, then use the scripts provided in the `deploy` folder to spin up a VM, and later tear it down. Once the VM is spun up, one needs to `ssh` into the VM and use the scripts in the relevant experiment directory. Once the experiment is finished, the VM can be torn down. Note that as mentioned before, some of the scripts might not work due to private dependencies. In such cases, use the scripts as reference.

⁵Time and cost estimate sheet: <https://tinyurl.com/52mwtccn>

A.4.1 Getting Started

Please see the artifact repository [Getting Started section of the README](#) for instructions on how to write a simple Cachew input data pipeline and execute it locally.

A.4.2 Reproducing Experiment Results

We provide scripts to automate the deployment, execution, and result plotting for the three key experiments listed in §A.1. See the [Artifact Evaluation section of the README](#) for instructions to run the scripts. Please follow the following steps for each experiment:

1. Deploy a VM for artifact evaluation using `deploy/deploy.sh <vm-name> <gpu-count>`
2. Use the `gcloud compute ssh <vm-name>` command to ssh into the VM
3. Use `cd ${HOME}/cachew_experiments/experiments/<ename>` where `<ename>` is the experiment name, and follow the README there and the associated scripts to run the experiments.
4. Use `gcloud compute scp` to collect whatever resource you find relevant after the experiment is done.
5. Exit the ssh session, and tear down the VM using the `deploy/terminate.sh` script.

For Figure 6a (compute curve) experiment, which required 4 GPUs, see the `experiments/autoscaling` directory. For Figure 8, see the `experiments/autocaching` directory. For Figure 11, see the `experiments/multi-tenancy` directory.

A.5 Evaluation and Expected Results

Each of the three experiments produces a plot which should be comparable with the associated plot in the paper.

A.5.1 Experiment Metrics

- *Figure 6a*: Epoch time in seconds, number of workers chosen by Cachew’s autoscale policy and number of workers chosen by Kubernetes HPA
- *Figure 8*: Batch time, Cachew’s autocache policy decisions
- *Figure 11*: Epoch time, autocache policy decision, autoscale policy decision

A.5.2 Expected Results and Possible Variations

- *Figure 6a*: Epoch time can vary depending on cloud conditions. Decay may or may not be more or less aggressive due to this. Consequently, autoscale decision might vary around 4 workers (at most ± 1 worker). While it is rare, it can happen that the Kubernetes HPA scaling also changes from one worker to two.
- *Figure 8*: Epoch times might vary due to cloud conditions. Shape of curves should still be the same (although *compute* might change as it depends heavily on GCS). Note that the points on the curves, and Cachew’s decisions are recorded separately (i.e. in different runs). Consequently, conditions might change, and metrics could potentially vary leading Cachew to make a seemingly ‘wrong’ decision at points where the three options have similar throughput. Otherwise, Cachew decision expected to follow lowest batch time option.
- *Figure 11*: Job 1’s first two epochs are not always expected to converge during autoscale phase (as Cachew prefers to move to next epoch in those compute modes), but it could happen at 3 workers (both epochs). Job 1’s third epoch expected to converge around 2 workers. Job 2 is expected to converge around 4 workers. Epoch times for Job 1 should be around [366s, 363s, 266s, 253s] while for Job 2 around 158s initially then around 129s in the later epochs. The expected sequence of execution modes for Job 1 is [PROFILE, PUT, GET] and for Job 2 is only GET. Both jobs can have epoch extensions towards the end of a run. A reasonable amount of variability in the worker count (± 1 worker) and epoch times is expected. This experiment is expected to be the relatively volatile, and emphasis should be placed on epoch time convergence and the autocache decisions. On rare occasions the autoscaling decisions can converge to a wrong scale due to the short and synthetic nature of the experiment and the implicit noise this causes on the scaling metrics. As this is a short job, Cachew cannot correct the scale in good time. We have also recently identified a bug caused by the merge with the recent TensorFlow 2.8 codebase which affects our cache store. More detailed information regarding this experiment’s expected outcome and variability can be found in its README.

A.5.3 Experiment customization

It should be possible to modify some of the parameters pertaining to each experiment. For instance, for the Figure 6a experiment, it is possible to change the number of workers across which clusters are deployed. These parameters can be changed in the experiment scripts themselves or for some as command line parameters. Please see the experiment scripts for further details.

CoVA: Exploiting Compressed-Domain Analysis to Accelerate Video Analytics

Jinwoo Hwang
KAIST

Minsu Kim
KAIST

Daeun Kim
KAIST

Seungho Nam
KAIST

Yoonsung Kim
KAIST

Dohee Kim
KAIST

Hardik Sharma
Google

Jongse Park
KAIST

Abstract

Modern retrospective analytics systems leverage cascade architecture to mitigate bottleneck for computing deep neural networks (DNNs). However, the existing cascades suffer from two limitations: (1) decoding bottleneck is either neglected or circumvented, paying significant compute and storage cost for pre-processing; and (2) the systems are specialized for temporal queries and lack spatial query support. This paper presents CoVA, a novel cascade architecture that splits the cascade computation between compressed domain and pixel domain to address the decoding bottleneck, supporting both temporal and spatial queries. CoVA cascades analysis into three major stages where the first two stages are performed in compressed domain, while the last one in pixel domain. First, CoVA detects occurrences of moving objects (called *blobs*) over a set of compressed frames (called *tracks*). Then, using the track results, CoVA prudently selects a minimal set of frames to obtain the label information and only decode them to compute the full DNNs, alleviating the decoding bottleneck. Lastly, CoVA associates tracks with labels to produce the final analysis results on which users can process both temporal and spatial queries. Our experiments demonstrate that CoVA offers $4.8\times$ throughput improvement over modern cascade systems, while imposing modest accuracy loss.

1 Introduction

Every day, a massive corpus of video data is produced, which is only growing (9.4 exabytes per day, as of 2021 [1]). Extracting insights and actionable semantics from the captured video can enable a variety of applications in healthcare, smart cities, security, customer behavior analysis, etc. Prior works [2–7] have built *retrospective analytics systems* that allow analysts to interactively query over a large corpus of accumulated video data stored in disk.

Modern retrospective analytics heavily rely on deep neural networks (DNNs). Although DNNs are effective, they come

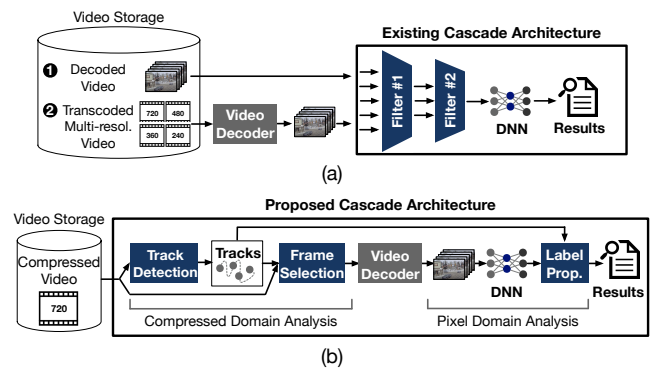


Figure 1: (a) Existing state-of-the-art cascade systems [2, 3], excluding video decoding from the end-to-end setting with two costly assumptions; (b) the proposed cascade architecture that addresses the decoding bottleneck and supports spatial queries, exploiting the compressed domain analysis.

at the cost of significant compute complexity, even for an image. Evidently, passing all the frames of a video through DNN inferencing is computationally prohibitive. To address this challenge, recent works [2–4, 6–13] have focused on *cascade* architectures. They stage processing as (relatively) inexpensive predicates to filter the incoming frames of video by trading analysis accuracy for higher throughput. As such, only a handful of frames arrive at the last stage that performs the full DNN inferencing.

While effectively resolving the DNN throughput bottleneck, the existing cascade systems have two limitations. First, as shown in Figure 1(a), these systems either ignore or sidestep a new bottleneck stage, *video decoding*, by making one of the two costly assumptions: (1) input video is decoded a priori and the raw frames are stored in storage [2, 3, 5, 7], or (2) input video is pre-transcoded and stored in multiple lower resolutions at ingest time to facilitate the query time decoding [4, 6]. However, in practice, decoding (or transcoding) the entire video corpus and storing the uncompressed (or

duplicate) data in disk is often infeasible due to the significant compute and storage cost.

Second, to achieve otherwise-unachievable throughput, the existing cascade systems often exclusively support *temporal* queries. More specifically, many cascade systems [2, 3, 5, 11] only support binary predicate query, which is to get timestamps of frames that contain the queried object. However, recent studies in video analytics [7, 15] propose *spatial* queries (e.g., car in upper right region) and demonstrate their usefulness, which cannot be supported by the current cascades.

To tackle the two limitations, this paper sets out to devise CoVA¹, an alternate cascade architecture. As illustrated in Figure 1(b), the key contribution of CoVA is to split cascade computation between compressed domain and uncompressed pixel domain, which collaboratively alleviate the decoding bottleneck at query time without requiring any pre-processing and support both temporal and spatial queries. To design this cascade architecture, we leverage the following two insights:

- (1) A small set of encoding metadata, commonly used by modern video codecs, provides noisy, yet rich, information to accurately locate potential objects and track them across frames in compressed video, while decoded pixel data is only necessary to classify objects.
- (2) Video analytics queries can be fulfilled by answering the following three questions: (1) where and when are interesting objects present in the video (i.e., spatiotemporal information); (2) what are the object classes (i.e., label information); and (3) what specific information do queries ask about these objects?

With these insights, CoVA divides video analytics over compressed footage into three major stages. The first stage (**Track Detection**) detects occurrences of moving objects (called *blobs*) over a collection of consecutive compressed frames (called *tracks*). To realize this objective, we devise a novel compressed-domain blob tracking technique, refitting a neural network based segmentation algorithm and a multiple object tracking algorithm, both of which are originally designed for pixel domain. Our second stage (**Frame Selection**) avoids decoding the whole track and selects a minimal set of frames that are representative and yet minimize the decoding load. CoVA passes only this subset through the full DNN object detection. The third stage (**Label Propagation**) takes the labels and the coordinates of the detected objects in the subset and uses spatiotemporal information from the first stage to propagate labels across all the frames of the track. Altogether, these approaches offer a novel cascade architecture that performs its first and second stages in the compressed domain, while the third stage is in the pixel domain.

Finally, the three stages produce a collection of analysis results for each frame, which include a list of present objects,

¹CoVA: Compressed Video Analytics.

their pixel coordinates, their labels (e.g., car), and all other properties associated with the objects (e.g., color). Note that the results are query-agnostic and not specific to a certain query. Therefore, CoVA runs the three stages only for the initial query and stores the analysis results along with the video in database. When other queries are requested over the same video in a future, CoVA simply retrieves the results and process the queries without reprocessing the video.

We prototype a CoVA system² on NVIDIA's streaming analytics framework, DeepStream [16]. We evaluate the effectiveness of CoVA using five video streams and four queries. Compared to existing cascade systems for query time retrospective analytics, CoVA offers 4.8× throughput improvement, while compromising only modest accuracy loss. We also show that CoVA is capable of serving spatial queries without having significant accuracy loss, compared to the full DNN analytics baseline.

Contributions. Our key contributions are as follows:

- We show that encoding metadata is sufficiently rich to identify objects of interest along with their spatiotemporal information for retrospective video analytics.
- To extract the spatiotemporal information, we devise a novel compressed-domain blob tracking technique, refitting the pixel-domain video segmentation and object tracking algorithms.
- We present the design of CoVA, a mixed-domain retrospective analytics system that leverages the track information to alleviate the decoding bottleneck, and support both temporal and spatial queries.
- Our experiment shows that CoVA offers significant throughput improvement over conventional cascade systems, while compromising modest accuracy loss.

2 Background and Motivation

CoVA aims to tackle limitations of existing retrospective analytics systems. Below, we first provide background on state-of-the-art retrospective analytics and discuss their limitations. We also discuss common compression mechanisms of modern video codecs, which drive the design of proposed techniques.

2.1 Retrospective Analytics

Modern retrospective analytics systems [2–8, 10–14] share two common properties: (1) heavy reliance on DNNs, and (2) cascade architecture to resolve the DNN compute bottleneck.

²Our prototype is available at <https://github.com/casys-kaist/CoVA>.

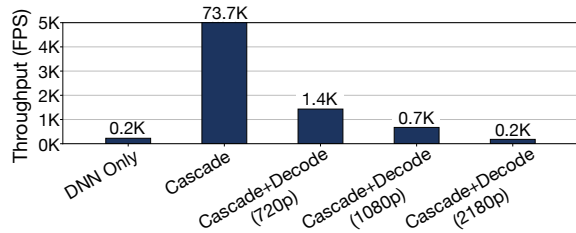


Figure 2: Throughput comparison among various system environments of cascade video analytics.

While they have these common properties, there are two different dimensions that categorize the instances of retrospective analytics systems.

Time of analysis – query time vs. ingest time. Retrospective analytics systems are categorized into two groups, depending on whether the analysis occurs at *query time* [2, 3, 11] or *ingest time* [4, 6, 8, 13]. While ingest time analysis leverages offline pre-processing to facilitate and expedite the query time analysis, it requires to scan the entire video data corpus and consume compute resources on it, even though a significant portion of the data is not queried. This approach is not only cost-inefficient but also environmentally suboptimal since it would consume a massive amount of energy for mostly unnecessary computations. In contrast, query time analysis performs the full analysis at query time without having any pre-processing. Therefore, it does not touch raw video data unless it is queried, which allows analysts to prevent the waste of resources. To this end, this work focuses on the query time analysis and aims to address its limitations.

Supported query – temporal vs. both temporal and spatial. Most, if not all, of query time cascade systems [2, 3, 11] limit the types of supported queries to be only the *temporal* ones and specialize the cascade stages for a specific temporal query to achieve high throughput. However, recent work [7] points out that *spatial* information can enable richer capabilities for video analytics. CoVA is a novel cascade architecture that leverages compressed-domain analysis to address both spatial and temporal queries.

2.2 Video Decoding: the New Bottleneck

Decoding for end-to-end cascade. With the volume of video data growing at an explosive rate, the use of compression is imperative to keep storage costs in check. Video codecs such as H.264 strike a balance between quality and storage size, being used as the de facto way of storing large corpus of video data. As such, the first step in an end-to-end system for processing video queries is to decode the video data before further processing. However, even with hardware-

acceleration for standard codecs baked-in to modern CPUs and GPUs, video decoding can be up to orders-of-magnitude slower than the capabilities of cascade systems to process raw video frames.

Bottleneck analysis. To quantify this bottleneck, we examine the performance impact of video decoding for an existing state-of-the-art cascade system, Tahoma [3], using NVIDIA RTX 3090 GPU, and present the results in Figure 2. The detailed methodology is provided in Section 8.1. The cascade system is effective in addressing the DNN-execution bottleneck and offers up to $327\times$ improvement in performance compared to a native DNN-only solution. However, even with decoding accelerator hardware NVDEC [17], the decoding throughput is significantly lower than the throughput of cascade system, which curtails most performance gains.

Further, as video resolution increases, the decoding throughput almost linearly decreases, exacerbating the decoding bottleneck. Considering the trend that even IoT devices such as surveillance cameras produce HD (1080p) or higher resolution video, we believe that this decoding bottleneck will become increasingly severe and significantly hinder the usefulness of video analytics in interactive applications. Motivated by these insights, the objective of CoVA cascade is to address the decoding bottleneck in query time retrospective analytics.

2.3 Block-based Video Coding

To alleviate the decoding bottleneck, CoVA leverages the unique characteristics of *block-based* compression, used in many modern video codecs. Below, we provide background on block-based compression and discuss opportunities that it opens for compressed-domain analysis.

Video codecs. Many video codecs, such as H.264, HEVC, VP8, VP9, and AV1, use block-based compression algorithm. In this paper, we primarily focus on the H.264 format since it is one of the most widely used codecs in various applications as of publication date [18]. However, CoVA is compatible with other block-based codecs since all of them compress video, generating the same set of metadata we use for compressed-domain analysis in CoVA.

Block-based compression. Block-based codecs compress (or encode) video frames by splitting each frame into a two-dimensional array of fixed sized blocks, called *macroblocks* (e.g., 16x16 pixels). There are three *macroblock types* – I, P, and B – depending on the way how the macroblocks are compressed. An I-macroblock is independently compressed, while P- and B-macroblocks are compressed referring to one and two other macroblocks, respectively. To maximize compression ratio, the codecs select dependent macroblocks for P and B-macroblocks with the highest similarity and store the spatial offsets as metadata called *motion vectors*. Depending

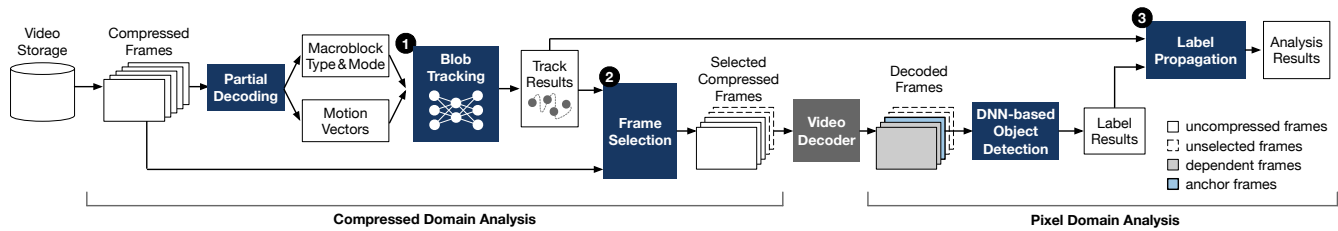


Figure 3: Overview of CoVA.

on the composition of macroblocks, frames are again categorized into three types, I, P, and B. An I-frame, also known as a keyframe, is only composed of I-macroblocks, while a P-frame contains I/P-macroblocks and a B-frame has all of the I/P/B-macroblocks.

To maximize the compression rate, codecs can *partition* macroblocks (e.g., 16x16) into smaller sub-macroblocks (e.g., 4x4). This optimization allows codecs to achieve a higher compression rate but at the expense of storing larger metadata. Modern codecs employ multiple *macroblock partitioning modes*. For instance, H.264 uses six modes from no partitioning (i.e., 1 macroblock of size 16x16) to 16-way partitioning (i.e., 16 sub-macroblocks of size 4x4).

CoVA leverages the insight that the encoding metadata – (1) macroblock types, (2) motion vectors, and (3) macroblock partitioning modes – in the compressed video is sufficiently rich to detect potential objects and track them across frames.

Compression rate optimization. Due to the higher compressibility, codecs tend to prefer P/B-macroblocks over I-macroblock. However, the preference for P/B macroblocks ends up creating long dependency chains among the macroblocks, which cause compression errors to propagate across the chains and hinder random access to frames in the video. To resolve the problems, the codecs insert I-frames at regular intervals, typically every 250 frames, to create independent sets of consecutive video frames, called Groups of Pictures (GoP). Within a GoP, the number of dependent frames that need to be decoded grows linearly, with zero for the first I-frame and maximum for the last frame.

CoVA exploits the inter-frame dependencies and object track information extracted from compressed-domain analysis to prudently select the frames with the least number of dependencies in each GoP that enable to identify all the objects present and minimize decoding effort.

3 Overview of CoVA

CoVA divides video analytics over compressed footage into three major stages, as illustrated in Figure 3.

1 First Stage: Track Detection. First, CoVA detects occurrences of moving objects over a collection of consecutive compressed frames, which we call tracks. The track detection stage further breaks down into two steps: (1) *blob detection*: CoVA *spatially* detects whether and where moving objects (called blobs) are present in each compressed frame; and (2) *blob tracking*: CoVA *temporally* associates the blobs across frames to identify unique blob tracks. For the blob detection, we devise a novel compressed-domain blob detection model, refitting a neural network architecture originally designed for pixel-domain video segmentation. The neural network only takes as input three encoding metadata commonly used by modern codecs, recognizes movements as masks, and spatially associates the masks clustered in a region as blobs. While the neural network architecture is fixed, CoVA trains the model individually for each video to learn the data-specific patterns of blobs and specialize for the target video. Finally, the found blobs are fed into the blob tracking step that employs an object tracking algorithm, SORT [19], which was also originally developed for pixel domain. Note that the blob track results still lack the object class labels.

2 Second Stage: Frame Selection. To attain the object labels for the detected blobs, CoVA needs to perform DNN-based object detection for the frames where tracks appear, which ordinarily require decoding all the frames. However, as frames on a track most likely contain the same object, it is enough to perform the object detection on a subset of the frames in the track, which we call *anchor frames*. Thus, CoVA only decodes frames required to decode the anchor frames, which improves the effective decoding throughput. The challenge is how to prudently select the anchor frames so as to minimize the decoding cost and at the same time acquire the accurate label information. We develop a frame selection algorithm that leverages a common property of video codecs where compressed frames are encoded in dependency chains. Thus, anchor frames are the ones that are located on the max-

imal number of tracks and at the same time have the short dependency chain with respect to the decoding algorithm. Note that while the anchor frames are the only ones that are inferred upon for object detection, all the frames in the track need to be labeled to handle various video analytics queries.

③ Third Stage: Label Propagation. In the third stage, CoVA takes the approximate positions of potential objects (or blobs) from the first stage and labels for the anchor frames from the second stage to temporally propagate the labels across all the frames of the tracks. To merge the spatial and temporal results, CoVA first spatially correlates blobs with objects on anchor frames using the intersection ratios of their bounding boxes. Then, CoVA uses the tracking information to identify the same objects across the frames and propagates the labels, while populating bounding boxes around the corresponding blobs in the temporally consecutive frames.

Finally, when a video passes through the three stages, CoVA produces a collection of analysis results for each frame, the examples of which are a list of present objects, their pixel coordinates, their labels (e.g., car), and all other properties associated with the objects (e.g., color). Note that the results are created only once when CoVA receives the initial query over a video and they are permanently associated with the video in the database. After then, analysts can use the same results to process various future queries without reprocessing the video.

4 Compressed Domain Blob Tracking

In this section, we describe the track detection mechanism that is the first stage of CoVA’s cascade architecture. Figure 4 depicts the overall workflow.

4.1 Learning to Detect Blobs

Limitations of existing compressed domain video processing techniques. Detecting objects or blobs from compressed video is a traditional research problem in the computer vision community [20–25]. However, the following two limitations prevent the simple adoption of these techniques. First, the techniques often require human-crafted parameters that need to be tuned for each input video, which makes automated analytics impossible. Secondly, the techniques are not sufficiently robust to be applied to arbitrary video data, producing inadequately accurate tracking results for video analytics. To overcome such limitations, recent works [26, 27] explored to use neural networks for vision tasks over compressed video. Unfortunately, we could not employ the neural networks for CoVA since they not only still require pixel-domain data for a subset of frames, but also offer insufficient throughput that is significantly lower than the decoder.

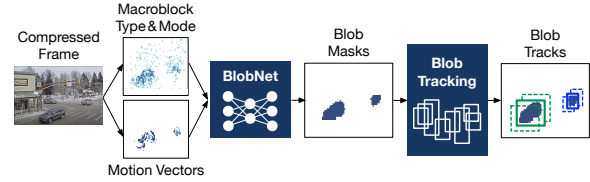


Figure 4: Track detection.

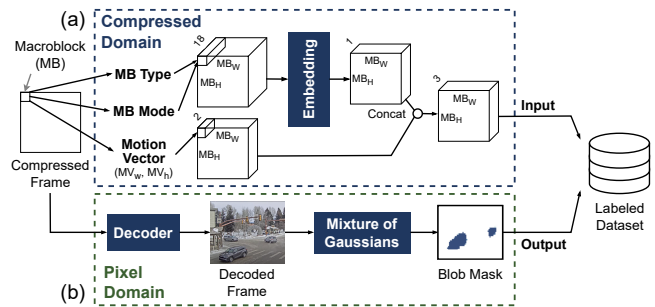


Figure 5: (a) Feature engineering that transforms three compression metadata into a tensor of input features; (b) labeled data collection using the Mixture of Gaussians (MoG) model.

Leveraging the similarity between video segmentation and blob detection. To address these limitations, we exploit an observation that blob detection using compression metadata is akin to the problem of the semantic image segmentation using pixel data. *Blob detection* task aims to find potential objects and their approximate position within video frames. *Image (or video) segmentation* task, on the other hand, aims to semantically split an image (or frames of a video) and classify each segment into one of the predetermined labels. When there are only two classes – blob and non-blob – the image segmentation task can be reduced to the approximate blob detection task. This observation allows us to tap into the vast range of techniques, including Deep Neural Network (DNN) based image and video segmentation, that can be geared towards compressed domain blob detection.

4.2 BlobNet

To this end, we devise a lightweight DNN-based blob detection model, called BlobNet, building upon the state-of-the-art Temp-UNet [28] model for video segmentation. Unlike the Temp-UNet model, which operates on pixel frames, BlobNet operates on compression metadata.

Feature engineering. Figure 5(a) depicts the feature engineering, which converts the three metadata into a tensor of input features. BlobNet takes the three types of metadata as input – macroblock types, macroblock partitioning modes, and motion vectors. To obtain the metadata, CoVA performs

only a few early stages of the decoding process required to extract metadata, called *partial decoding*. CoVA encodes the first two metadata, macroblock types and partitioning modes, by mapping each of their combinations into a one-hot vector (e.g., total 12 combinations for H.264). These one-hot vectors are fed into an embedding layer, which converts each one-hot vector into a scalar weight value. This weight value is concatenated to the motion vector (MV_w, MV_h) for each macroblock, which finally results in a 3D tensor ($MB_W \times MB_H \times 3$). CoVA temporally stacks these tensors from consecutive frames and constructs a 4D tensor, which is the input for BlobNet.

BlobNet architecture. Similar to the architecture of TempUNet³, BlobNet has three major components: (1) **encoder** that extracts the presence and approximate location of blobs from noisy metadata; (2) **decoder** that reconstructs the shapes of blobs from the blob presences; (3) **skip connections** that offer spatial information to the decoder for assisting the shape reconstruction process. While this overall composition is the same as that of TempUNet architecture, we maximally reduce the depth of encoder and decoder modules such that the resulting model still offers high accuracy while maximizing the inference throughput.

Video-specialized model training. Pixel video segmentation models typically train once during a training phase, followed by inference on unseen video data. However, CoVA trains BlobNet at query time for every video data to specialize the model for the specific data. This design choice is derived from our empirical observation that without such model specialization, the model cannot capture the variations of data-specific encoding parameters and fails to reach sufficient accuracy. Note that once training is completed for a video data, no further training is required for additional video if the video is recorded from the same angle of view with the trained one. We empirically observe that $\approx 3\%$ of the video is sufficient to train the model for the evaluated video (see Table 2). The training process, including data collection and training, takes only a few minutes, which can be amortized for multiple queries on the same video data. Such training cost amortization is inspired by existing query-time cascade systems [2, 3, 6, 8] that train specialized neural networks for each video.

Labeled data collection for supervised learning. As CoVA aims for large-scale video analytics, manually labeling the video data is infeasible. As such, CoVA needs a method to automatically label the video data. Similar to prior works [2, 3, 8, 29], using pixel domain object detection is a possible option. However, object detection models are not only computationally expensive but also produces labels for non-moving objects, which should not be used to train BlobNet, designed to detect only moving objects. Instead, we exploit the conventional Mixture of Gaussians (MoG) based background

³We omit the detailed architecture and refer to the paper [28].

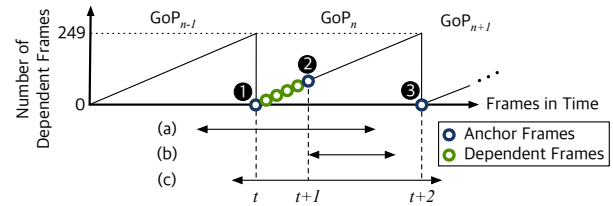


Figure 6: Example scenario of track-aware frame selection.

subtraction technique since it is lightweight and only looks for the moving objects.

4.3 Tracking Blobs

Blob detection results. The output of BlobNet is merely a collection of 1’s on the resulting bitmap, which lacks the notion of objects. CoVA uses connected-component labeling algorithm to uniquely identify the interesting regions in compressed frames as potential objects, called *blobs*. Once the blob identification process is completed, CoVA obtains the spatial information of blobs on each frame. However, the blobs existing across consecutive frames are not yet temporally associated with each other, which necessitates the next stage of CoVA, blob tracking.

SORT-based blob tracking. The end objective of blob tracking in compressed domain is to minimize the number of frames to be decoded to mitigate the decoding bottleneck. Hence, the tracking algorithm must (1) offer high throughput that significantly outperforms the decoder throughput, (2) while accurately tracking the inter-frame blobs to minimize the accuracy loss at the label propagation stage. We extensively explore existing object tracking techniques in pixel domain [19, 30–36], and choose the SORT object tracking algorithm [19], which satisfies the above two requirements. SORT offers the near-best tracking accuracy among the state-of-the-art tracking techniques and keeps the computation lightweight by exploiting conventional optimization algorithms, Kalman filter and Hungarian assignment.

5 Track-aware Frame Selection

Leveraging the track information, CoVA prudently select a small subset of frames to decode, called *anchor frames*, so as to maximize the decoding throughput. The key idea behind the anchor frame selection algorithm is to pick the ones that require to decode the least number of frames and thus maximize the *effective* decoding throughput.

Dependency between compressed frames. As described in Section 2.3, block-based compression uses a combination of (1) independent frames that are self-contained (i.e., I-frame), and (2) dependent frames (i.e., P/B-frames) that depend on

Input : efs: compressed frames in a GoP_{*t*}
tracks: blob tracks that maintain across GoPs

Output : dfs: compressed frames chosen to be decoded
afs: anchor frames

```

1 cur_tracks = tracks that terminate in GoPt
2   with no anchor frames assigned
3 dfs = afs = ∅
4 if cur_tracks ≠ ∅ then
5   start_timestamps = sorted(cur_tracks.starts())
6   end_timestamps = sorted(cur_tracks.ends())
7   sidx = eidx = 0
8   for ef in efs do
9     while start_timestamps[sidx] == ef.timestamp do
10      candidate_af = ef
11      sidx = sidx + 1
12    end
13    while end_timestamps[eidx] == ef.timestamp do
14      afs.add(candidate_af)
15      dfs.add_dependants(candidate_af, efs)
16      eidx = eidx + 1
17    end
18  end
19 end
20 dfs.output()
21 afs.output()

```

Algorithm 1: Track-aware frame selection algorithm.

either preceding frames, subsequent frames, or both. Due to the presence of P-frame and B-frame within a GoP, the number of dependent frames that need to be decoded to fully decode a frame follows a saw-tooth structure, as depicted in Figure 6. The number of dependent frames is zero for I-frame at a GoP boundary and grows linearly until it resets to zero at the end of GoP⁴.

Selecting anchor frames for decoding. To minimize the decoding load, we leverage two insights: (1) CoVA can find the consecutive frames where an object keeps appearing in the video, and (2) the computations load to decode a frame is proportional to its number of dependent frames. Within each GoP, CoVA identifies a set of anchor frames, which can identify all objects present in the GoP and perform the least computation for decoding, by minimizing the number of dependent frames. The selected anchor frames are the only ones that are passed to the DNN object detector to produce the label information.

Example. Figure 6 presents an example where CoVA identifies three unique objects, (a), (b), and (c), as well as the range of frames where each object stays in the video. In this example, the best choice of anchor frame would be Frame ② since (1) all the objects are present in Frame ②, and (2) Frame ② has the least number of dependent frames among frames where all the objects are present.

⁴For brevity, we simplify Figure 6 by only visualizing dependency chains for P-frames since the number of dependent frames for B-frames is similar to that of the nearby P-frames.

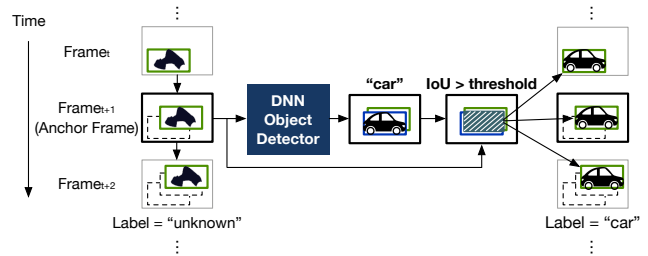


Figure 7: Label propagation.

Algorithm. Algorithm 1 describes the frame selection algorithm in detail. *Line 1*: When a GoP arrives at the frame filtering, to select the anchor frames, CoVA only considers tracks that (1) terminate in that particular GoP and (2) do not have any anchor frames yet (e.g., object (a)/(b) at time *t*). *Line 9*: Then, as CoVA visits frames in order, it first checks if a track starts appearing in the visiting frame. *Line 10*: If it does, the visiting frame is marked as “candidate” anchor frame (e.g., Frame ① at *t*). Later on, if a new track starts appearing in a successive frame, the frame becomes the new candidate (e.g., Frame ② at *t*+1). *Line 14–15*: When a track ends, CoVA adds the current candidate frame into the anchor frame list (e.g., Frame ②) and inserts all the dependent frames into the dependent frame list (e.g., all frames between Frame ① and Frame ②). The intuition behind this algorithm is that, if a track started but did not terminate, any frame in between can be an anchor frame. However, when a track ends, an anchor frame for the track must be selected, because otherwise, we may not have any anchor frame for the terminating track.

6 Label Propagation

In the last stage, CoVA takes the blob tracks and labels for the anchor frames to temporally propagate the labels across all the frames on the tracks. Figure 7 illustrates the example workflow of label propagation. When the selected anchor frames and their dependent frames are decoded, CoVA takes only anchor frames to perform the DNN object detection and obtain the labels (e.g., “car”) as well as their spatial information. To associate the labels with blobs, CoVA first spatially correlates blobs with the detected objects using the intersection over union (IoU) between their bounding boxes (e.g., bounding boxes of blobs and detected objects are denoted using green and blue boxes, respectively). When the IoU is larger than a threshold, CoVA associates the detected objects with blobs and propagates the labels to all frames in the tracks.

Multiple-objects overlapping problem. One challenge with the label propagation mechanism is that when BlobNet fails to separately identify multiple objects clustered together and creates a large single blob, CoVA cannot correctly propagate the multiple labels. To overcome the challenge, we prepend

an additional step to the label propagation. When a multitude of detected objects are spatially overlapped with a single blob, CoVA splits the blob into multiple blobs, proportionally projecting the locations of objects in the anchor frame to the blob. The proportional projection is also applied to other frames in the same track, populating multiple tracks from a single track. This way, CoVA is able to propagate the multiple labels to the separated tracks, instead of giving a single erroneous label to the clustered objects.

Static object handling mechanism. As CoVA relies on the compressed domain analysis to detect blobs, it is impossible to detect static objects from the compression metadata. Therefore, our BlobNet focuses on detecting moving objects, intentionally excluding the static object information from the training data through the use of MoG. However, CoVA still performs full-fledged object detections on anchor frames. Therefore, the static objects can be detected at least on the anchor frames. As the static objects stay still at the same location across subsequent anchor frames, CoVA is able to associate them as the same object and produce the corresponding track.

7 Implementation

System architecture and constituent software modules. We prototype a CoVA system using DeepStream, which is built upon GStreamer, for constructing the skeleton pipeline of video analytics. As described in Section 4, the initial stage of CoVA is the partial decoding, which extracts the metadata. Hardware-accelerated decoder (e.g., NVDEC) does not support partial decoding and only generates the fully decoded frames. Thus, we modify an open-source video codec, libavcodec, such that it only produces the three types of metadata. In addition, CoVA performs two neural network inferences, one for the blob detection and the other for the full DNN inference (YOLOv4). We use on a TensorRT-based DNN inference plugin on DeepStream, nvinfer [37].

Parallelization in CoVA. Our prototype system distributes the computations of pipeline stages over CPU and GPU, while exploiting their parallelism. Initially, CoVA scans the entire video and splits it into chunks at the I-frame boundaries to parallelize the computation on CPU threads. This scanning takes just a few seconds even for hours of video data, which imposes negligible overhead. Such parallelization results in cutting tracks at the chunk boundaries, but its impact on accuracy is negligible since there are only a few dozens of chunks. For a chunk, the first two stages, track detection and frame selection, should be pipelined in the same thread since these algorithms rely on the temporal dependencies of frames. For object detection, anchor frames are independently computed, which can be fully parallelized. Therefore, CoVA maintains only a single thread for object detection and anchor frames from different chunks are batched together for inference.

Table 1: Descriptions of example video analytics queries.

Query	Abbr.	Description	Metric
Binary Predicate	BP	Return frames where querying object appears	Accuracy
Count	CNT	Return the average count of querying object in frames	Absolute Error
Local Binary Predicate	LBP	Return frames where querying object appears in a certain region of frames	Accuracy
Local Count	LCNT	Return the average count of querying object in a certain region of frames	Absolute Error

8 Evaluation

8.1 Methodology

Queries. To demonstrate the effectiveness of CoVA, we evaluate four example queries, two queries widely used in prior work [2, 3, 8], and their spatial variants supported by CoVA. Table 1 reports the list of evaluated queries with their descriptions and accuracy metrics:

- (1) Binary Predicate.** Binary predicate (BP) query finds frames where queried objects appear. Collecting frames with queried objects is an initial step of advanced analysis, which makes BP an important query for evaluation despite the simplicity. Many retrospective analytics systems evaluate their solutions only using the BP query [2, 3].
- (2) Count.** The count (CNT) query is introduced by a prior work, BlazeIt [8], which is an aggregate query that counts the number of queried objects appearing in the whole video. As the aggregated count is largely dependent on the length of each dataset, the number is normalized by dividing it by the number of frame counts.
- (3) Local Binary Predicate and Local Count.** The local binary predicate (LBP) and local count (LCNT) queries are spatial variants of BP and CNT queries, respectively; however, the only difference is that they exclusively look for objects located in a certain region of interest. For instance, users can query northbound traffic in highway monitoring video by annotating the corresponding region of video as “northbound”. Serving these queries not only requires the temporal query results, but also needs spatial information to determine the object locations.

Metrics. Table 1 also reports metrics used for each query. We use the same metrics that prior works use to evaluate their

Table 2: Descriptions of video datasets, queried objects, ground truth results, and region of interest used for spatial queries. Note that we use the Yolov4 DNN model applied frame-by-frame to the original video to get ground truth.

Video Name	Num of Frames	Length	Object in Interest	Object Occupancy	Object Count	Local Occupancy	Local Count	Region of Interest
amsterdam [38]	3,580K	33H	Car	70.07%	1.40	29.05%	0.44	Lower Right
archie [8]	3,567K	33H	Bus	10.48%	0.17	6.63%	0.11	Upper Left
jackson [39]	2,921K	27H	Car	31.91%	0.56	18.28%	0.29	Lower Left
shinjuku [40]	1,782K	16H	Car	82.29%	2.19	19.91%	0.38	Lower Left
taipei [41]	3,564K	33H	Car	84.48%	5.03	22.16%	0.64	Lower Right

solutions. For BP and LBP, as in prior works [2, 3], we use *accuracy*, which is a traditional metric for binary classification that evaluates how many observations, both positive and negative, are correctly classified. Similarly, for CNT and LCNT, we use *absolute error* as used in Blazelt [8].

Datasets. Table 2 reports the video datasets used for the evaluation. Taking a similar approach with prior works [2, 3, 6–9, 42], we collect the video datasets from YouTube live streams [38–41]. They are recorded from statically installed cameras, which is a widely used setup in various applications domains such as traffic monitoring [43–45], security [46, 47], surveillance [48], and healthcare [49]. Video contents involve various kinds of scenarios, which include traffic circle, highway, harbor, city streets, and park. As the datasets have different resolutions ranged from 720p to 2160p, we transcode them to 720p and evaluate the throughput and accuracy for ease of comparison. Note that higher resolutions (e.g., 2160p) create more severe decoding bottleneck, so using them would be favorable to CoVA, producing higher throughput gains and therefore, to be conservative, we choose to use 720p for all video datasets. The rightmost five columns report the ground truth results for the four queries and the region of interest that spatial queries focus on. Getting the ground truth results by manually labeling the hours of video data is infeasible, so we apply a full DNN model (YOLOv4) to the entirety of video in a frame-by-frame manner.

Hardware specifications. Our CoVA prototype is built on a server with two 16-core Intel Xeon Gold 6226R CPU (2.9 GHz), 192 GB of DRAM, and an NVIDIA RTX 3090 GPU (24 GB GDDR6 DRAM). We turn off hyperthreading to avoid interference among threads.

Decoder. For all experiments, we use NVIDIA’s hardware accelerated decoder, NVDEC, for both baseline and CoVA systems to make a fair comparison. We choose not to use the CPU decoder, libavcodec, since it shows lower throughput than NVDEC even with 32-core parallelization.

Baseline cascade system. As the baseline, we use existing cascade systems for query time retrospective analytics. As discussed in Section 2, cascade systems such as Tahoma [3]

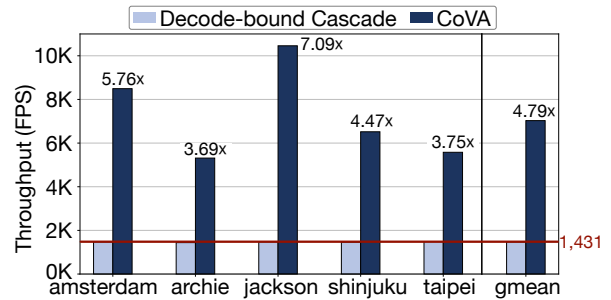


Figure 8: End-to-end system throughput of the baseline decode-bound cascade and CoVA. The throughput of decode-bound cascade is equivalent to the throughput of NVDEC (i.e., 1,431 FPS), which is marked with a red line.

are significantly bottlenecked by video decoding. Therefore, for a conservative comparison with these decode-bound cascade systems, we assume that the cascade systems are only bottlenecked by the decoder, not by any other stages. With this assumption, the throughput of cascade systems is equivalent to the decoder throughput. We refer to this baseline as *decode-bound cascade* in this paper.

8.2 Performance Implication of CoVA

Throughput improvement. Figure 8 compares the end-to-end system throughput of the baseline decode-bound cascade system and CoVA across five video datasets. CoVA achieves on average 4.8× throughput improvement, which ranges from 3.7× for archie to 7.1× for jackson. The significant speedup shows that CoVA effectively pushes a large proportion of analysis to the compressed domain, unclogging the decoding bottleneck that prevents the existing cascades to achieve beyond the constant NVDEC throughput. The results also suggest that depending on the datasets, CoVA sees different speedups. The datasets, jackson and amsterdam, see relatively larger gains, while archie and taipei datasets show lower benefits. These gaps can be attributed to the unique content properties of each evaluated video dataset that deliver varying throughput for the

Table 3: (1) Filtration rate at decoder stage (decode filtration rate) and (2) filtration rate at DNN inference stage (inference filtration rate).

Dataset	Decode Filtration Rate (%)	Inference Filtration Rate (%)
amsterdam	87.16	99.60
archie	72.94	99.15
jackson	94.81	99.79
shinjuku	77.18	99.26
taipei	74.03	99.81

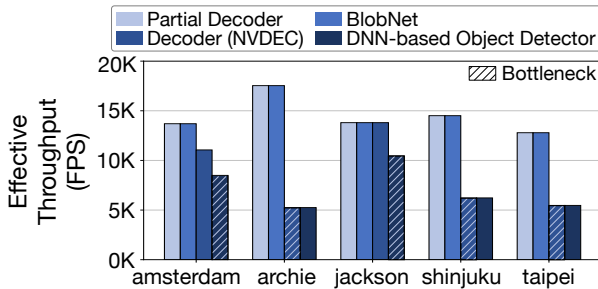


Figure 9: Effective throughput of CoVA stages. The lowest bar represents the bottleneck of CoVA pipeline, which is marked with hatching lines.

CoVA pipeline stages, which eventually engenders a different bottleneck point. To better understand the throughput implication of these stages, we delve into the interplay of algorithms and system in the CoVA pipeline below.

Effectiveness of frame selection. Frame selection is the key to alleviate the decoding bottleneck since it determines the computational load for decoder. Table 3 reports the filtration rates at decoding stage (decode filtration rate) and DNN inference stage (inference filtration rate). The decode filtration rate is calculated based on the number of decoded frames that include both anchor frames and their dependent frames, while the inference filtration rate only considers the anchor frames that are passed to the DNN object detection stage. Intuitively, various semantics of datasets cause different filtration rates. If video contains many objects having lots of motions, blob tracking would produce numerous tracks, which would require many anchor and dependent frames to proceed to the decoder. For crowded video streams such as archie, CoVA sees lower decode filtration rate of 72.94%, while the uncongested ones like jackson capture less activity and provide higher decode filtration rate of 94.81%. Across all datasets, CoVA filters out over 73% to deliver over $3.7\times$ ($=100/(100-73)$) throughput boost for decoder. At the same time, the inference filtration rate closely reaches 100%, which addresses the DNN bottleneck since the object detector only sees a handful of frames.

Bottleneck analysis. To understand the throughput variation

Table 4: Accuracy results of the four evaluated queries for the video datasets. The acronyms for accuracy metric are specified below.

Dataset	Object	BP (ACC)	CNT (AE)	LBP (ACC)	LCNT (AE)
amsterdam	Car	85.79	0.15	81.61	0.09
archie	Bus	86.96	0.04	90.06	0.01
jackson	Car	86.13	0.10	92.01	0.05
shinjuku	Car	90.15	0.30	91.31	0.05
taipei	Car	87.74	1.10	83.98	0.37
average	-	87.34	N/A	87.69	N/A

* ACC: Accuracy (%), AE: Absolute Error

of CoVA stages across different datasets, we measure the performance of individual stages. Figure 9 reports the effective throughput of each stage by starting from the first partial decoding stage and adding successive stages one by one to the system. The *effective* throughput is defined as the product of the absolute throughput of stage and the accumulated filtration rates. Note that since we measure the throughput from the pipelined system, the effective throughput of a stage cannot be larger than that of the previous stage. The results suggest that different datasets experience bottleneck at different stages. The datasets that attain lower decode filtration rate than the others (i.e., archie, shinjuku, and taipei) are still bottlenecked at the decoder, while the other two datasets are bounded by the DNN object detector. We observe that the inference of BlobNet never becomes a bottleneck and always matches the throughput of the preceding partial decoding stage.

8.3 Accuracy Implication of CoVA

Table 4 reports the accuracy results of evaluated queries. For the BP query, CoVA achieves on average 87.3% accuracy. For the CNT query, CoVA experiences absolute errors from 0.04 (archie) to 1.10 (taipei), respectively. For spatial queries (LBP and LCNT), we do not observe a noticeable difference in accuracy with the temporal queries. The lack of difference is intuitive since CoVA processes the spatial queries by simply restricting the focus of analysis to a certain region of frames. Therefore, the results of spatial variants are merely a subset of temporal query results.

The results show that the approximate nature of compressed domain analysis introduces accuracy loss. However, we argue that such modest level of accuracy degradation (10~20%) is tolerable to retrospective video analytics, which aims to process large corpus of video data interactively at query time. The video analytics also inherently produce approximate results due to the nature of noisy analog video data and predictive object detection models. Moreover, our accuracy results are conservatively calculated by treating the YOLOv4 detection results as ground truth and marking the CoVA results as error

Table 5: Raw throughput of four different video codecs on the libavcodec and NVDEC decoders.

Codec	Full Decoding (FPS)		Partial Decoding
	NVDEC	libavcodec	(FPS)
VP8	1,590	1,802	32,774
H.264	1,431	1,230	16,761
VP9	3,249	1,179	35,349
H.265	3,888	2,026	25,862

if they do not match. However, we empirically observe that there are many cases where YOLOv4 misses small objects when the objects are faraway from the shooting point, while CoVA can correctly detect them by successfully tracking blobs even for the small objects and propagating the correct labels to the tracks. In this case, the correct results of CoVA would be marked as false positives due to the erroneous ground truth.

Discussion. As discussed above, approximation is fundamentally inevitable for video analytics, because even the best effort results are still imperfect. Thus, our goal in designing CoVA is to achieve *acceptable* approximation accuracy loss for video analytics. According to a study [50], the level of acceptable approximation accuracy loss is higher when the users consider contexts such as application purpose and cost. We believe that CoVA could be a useful tool where analysts can quickly and cost-efficiently extract high-level insights from a large corpus of videos. For instance, consider an application that monitors traffic in a harbor in Amsterdam (see Table 2). For binary predicate query, it suffers from 15% accuracy loss. However, CoVA does not miss the cars completely from the video in most cases since the cars stay in the video for at least several tens of frames (only 2% of cars are eventually missed). Hence, if analysts merely wanted to estimate traffic, CoVA would be able to offer sufficiently precise results. We also believe that if an application requires more accurate results, CoVA could serve as an initial scanning tool that quickly identifies “worth-to-be-further-analyzed” video clips.

8.4 Sensitivity Study

Implication of video codecs. We implement the CoVA system based on H.264, one of the most widely used video codecs. However, to demonstrate applicability of CoVA to other block-based compression standards, we take three alternatives, VP8, VP9, and HEVC (i.e., H.265), and develop metadata extraction in their partial decoding implementations. Table 5 reports throughput results when using the four different codecs with 720p videos and 32 cores. The throughput of NVDEC for the four codecs ranges from 1,431 FPS to 3,888 FPS, which is significantly lower than the effective throughput of existing cascade systems and thus our problem statement regarding decoding bottleneck still holds true. In addition, we observe

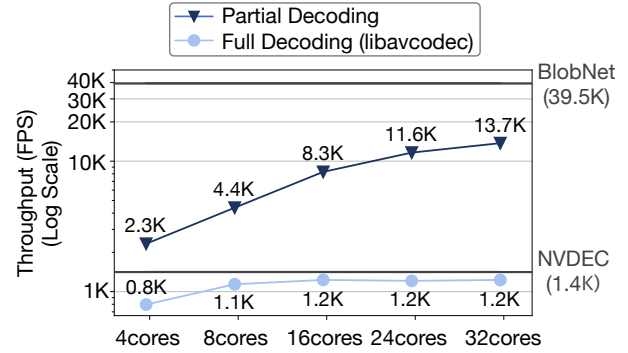


Figure 10: Throughput of partial and full video decoding (libavcodec) on CPU, as the number of cores changes from 4 to 32. For comparison, we also report the throughput of BlobNet and NVDEC, while they have constant throughput since they run on GPU.

that for all codecs, the full decoding throughput in both software and hardware significantly falls short of throughput of the partial decoding. This throughput gap allows CoVA to construct a cascade architecture that enables blob tracking to run at a higher throughput than the vanilla decoder and effectively lowers the full decode workload.

Implication of CPU parallelism. To further analyze the scalability of our parallelization scheme, Figure 10 compares the throughput of partial and full decoding as we parallelize them using the varied number of cores from 4 to 32. We also show the throughput of BlobNet and NVDEC for comparison. Note that these results are averaged across the datasets. The results show that the parallelized partial decoder not only scales significantly better than the full decoder when using the same number of cores (i.e., $1.5\times$ vs. $5.9\times$), but also largely outperforms the throughput of NVDEC. Currently, we use all the available cores (32) for partial decoding to optimize for throughput. However, one may be able to revise the objective function such that it also takes into account resource utilization and energy efficiency, which we leave as a future work.

9 Related Work

A growing body of literature [2–9, 14, 15, 29, 51–57] aims to address the computational challenges in video analytics. CoVA differentiates itself by addressing video decoding bottleneck, exploiting compressed-domain analysis. Further, CoVA does not require pre-processing, transcoding, or profiling to obtain the benefits.

Cascade architectures for binary predicate queries. No-Scope [2] and Lu et al. [5] use a series of approximate pixel-domain filtering stages to build their cascade. Tahoma [3] and Shen et al. [29] use multiple pipelined neural networks to build their cascade architecture. BlazeIt [8] builds on top

of NoScope to support Aggregate and Limit Queries. All five works aim to increase the effective throughput of the system for raw video frames by filtering a majority of the frames using pixel-domain operators. Alternatively, Thia [51] splits up the DNN-inference model using exit points for early termination, similar to the stages of cascade architecture. In contrast, CoVA splits the cascade computation between compressed domain and pixel domain to alleviate the decoding bottleneck.

Spatial queries for video analytics. An emerging class of video analytics systems aim to enable queries based on spatial relationship between labeled objects. Koudas et al. [7] accelerate spatial queries using separate stages for inexpensive DNN-based classification followed by expensive DNN-based object detection. TASM [15] dynamically adapts the layout of tiles, which partition compressed video frames, based on the spatial location of objects to improve performance. Unlike the above works, CoVA uses compressed domain blob tracking to accelerate spatial queries. Unlike TASM, CoVA does not need to update the compression to gain performance benefits.

Storage-accuracy trade-off for decoding bottleneck. VStore [4] uses a search space of fidelity and encoding/decoding knobs (frame sampling rate, resolution, etc) to optimize for query performance and storage cost. SMOL [6] jointly optimizes complexity of the reference DNN for inference and the resolution of data (360p, 720p, etc), for accuracy-performance trade-off. VSS [52] proposes optimizations for video storage to yield higher read rates and compression ratios. CoVA takes an orthogonal approach of performing approximate blob tracking using compression metadata at *query time*. Nevertheless, CoVA is complementary to the above approaches.

Ingest time analysis. Focus [9] generates approximate labels using an inexpensive DNN and Boggart [53] tracks objects at ingest time to generate additional metadata. At query time, both Focus and Boggart use the stored metadata to yield improved performance. Scanner [54] identifies sampling frames offline for pixel domain analysis and skips decoding for all other frames. In contrast, CoVA does not require additional metadata and can operate on standard video compression formats. VideoStorm [55] uses offline profiling data for dynamic load balancing and Chameleon [14] uses inexpensive online profiling to improve accuracy-resource tradeoff at query time. These two profiling approaches are orthogonal and complementary to compressed-domain query processing in CoVA.

Compressed domain object detection. Many prior works [23, 25, 58, 59] have proposed object detection from compression metadata using classical approaches such as pre-defined kernels [24] and statistical models [20, 60, 61]. Further, the prior works impose restrictions on the compression-time parameters (e.g., 4 frames per GoP), which limit their applicability [20, 23–25]. Liu et al. [62], Wang et al. [27], and Wu et al. [26] employ DNNs to detect moving objects using *both* pixel and compressed domain data, training a single

model for all datasets. BlobNet differs from prior works in the following aspects: (1) BlobNet does not require any pixel data; (2) BlobNet does not impose restrictions on the compression parameters; and (3) BlobNet is trained for given video to compensate the accuracy.

10 Conclusion

Existing cascade systems for video analytics assume to pay significant compute and storage cost for addressing the decoding bottleneck. Further, the systems are specialized for temporal query to achieve otherwise-unachievable throughput. To tackle the two limitations, this paper proposes CoVA, which splits cascade computation between compressed and uncompressed pixel domain. Leveraging the unique characteristics of video analytics and video compression algorithm, CoVA effectively unclogs the decoding bottleneck while additionally supporting spatial queries. Our experiments demonstrate that CoVA reduces the decoding workload by 83.6% and offers $4.8\times$ system speedup compared to state-of-the-art query-time retrospective video analytics systems, while compromising modest accuracy.

11 Acknowledgements

We thank the anonymous reviewers and our shepherd for their comments and feedback. This work was supported by National Research Foundation of Korea (NRF-2020R1A2C1103088) and Information Technology Research Center (ITRC) support program (IITP-2022-2020-0-01795), both of which are funded by the Ministry of Science and ICT, Korea. This work was also partly supported by Samsung Electronics Co., Ltd.

References

- [1] Mark Nowell. Cisco VNI Forecast update. https://www.ieee802.org/3/ad_hoc/bwa2/public/calls/19_0624/nowell_bwa_01_190624.pdf, 2021.
- [2] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. NoScope: Optimizing Neural Network Queries over Video at Scale. In *PVLDB*, 2017.
- [3] Michael R Anderson, Michael Cafarella, German Ros, and Thomas F Wenisch. Physical Representation-Based Predicate Optimization for a Visual Analytics Database. In *ICDE*, 2019.
- [4] Tiantu Xu, Luis Materon Botelho, and Felix Xiaozhu Lin. VStore: A Data Store for Analytics on Large Videos. In *EuroSys*, 2019.

- [5] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, , and Surajit Chaudhuri. Accelerating Machine Learning Inference with Probabilistic Predicates. In *SIGMOD*, 2018.
- [6] Daniel Kang, Ankit Mathur, Teja Veeramacheneni, Peter Bailis, and Matei Zaharia. Jointly Optimizing Preprocessing and Inference for DNN-Based Visual Analytics. In *PVLDB*, 2020.
- [7] Nick Koudas, Raymond Li, and Ioannis Xarchakos. Video Monitoring Queries. In *ICDE*, 2020.
- [8] Daniel Kang, Peter Bailis, and Matei Zaharia. BlazeIt: Optimizing Declarative Aggregation and Limit Queries for Neural Network-Based Video Analytics. In *PVLDB*, 2019.
- [9] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *OSDI*, 2018.
- [10] Ioannis Xarchakos and Nick Koudas. SVQ: Streaming Video Queries. In *SIGMOD*, 2019.
- [11] Jingjing Wang and Magdalena Balazinska. Deluceva: Delta-Based Neural Network Inference for Fast Video Analytics. In *SSDBM*, 2020.
- [12] Yuhao Zhang and Arun Kumar. Panorama: A Data System for Unbounded Vocabulary Querying over Video. In *PVLDB*, 2020.
- [13] Favyen Bastan, Oscar Moll, and Sam Madden. Vaas: Video Analytics At Scale. In *PVLDB*, 2020.
- [14] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: Scalable Adaptation of Video Analytics. In *SIGCOMM*, 2018.
- [15] Maureen Daum, Brandon Haynes, Dong He, Amrita Mazumdar, and Magdalena Balazinska. TASM: A Tile-Based Storage Manager for Video Analytics. In *ICDE*, 2021.
- [16] NVIDIA. DeepStream SDK. <https://developer.nvidia.com/deepstream-sdk>, 2021.
- [17] NVIDIA. Video Codec SDK. <https://developer.nvidia.com/nvidia-video-codec-sdk>, 2021.
- [18] Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the H.264/AVC Video Coding Standard. *TCSVT*, 2003.
- [19] Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos, and Ben Upcroft. Simple online and realtime tracking. In *ICIP*, 2016.
- [20] Mohammadsadegh Alizadeh and Mohammad Sharifkhani. Compressed Domain Moving Object Detection Based on CRF. *TCSVT*, 2020.
- [21] Wei Zeng, Jun Du, Wen Gao, and Qingming Huang. Robust Moving Object Segmentation on H.264/AVC Compressed Video Using the Block-Based MRF Model. *Real-Time Imaging*, 2005.
- [22] R. Babu, Kalpathi Ramakrishnan, and S.H. Srinivasan. Video Object Segmentation: A Compressed Domain Approach. *TCSVT*, 2004.
- [23] Marcus Laumer, Peter Amon, Andreas Hutter, and André Kaup. Moving Object Detection in the H.264/AVC Compressed Domain. *APSIPA*, 2016.
- [24] Chris Poppe, Sarah De Bruyne, Tom Paridaens, Peter Lambert, and Rik Van de Walle. Moving Object Detection in the H.264/AVC Compressed Domain for Video Surveillance Applications. *Journal of Visual Communication and Image Representation*, 2009.
- [25] Dien Van Nguyen and Jaehyuk Choi. Toward Scalable Video Analytics Using Compressed-Domain Features at the Edge. *Applied Sciences*, 2020.
- [26] Chao-Yuan Wu, Manzil Zaheer, Hexiang Hu, R Manmatha, Alexander J Smola, and Philipp Krähenbühl. Compressed Video Action Recognition. In *CVPR*, 2018.
- [27] Shiyao Wang, Hongchao Lu, Pavel Dmitriev, and Zhi-dong Deng. Fast Object Detection in Compressed Video. In *ICCV*, 2019.
- [28] Radu Sibeichi, Olaf Booij, Nora Baka, and Peter Bloem. Exploiting Temporality for Semi-Supervised Video Segmentation. In *ICCV*, 2019.
- [29] Haichen Shen, Seungyeop Han, Matthai Philipose, and Arvind Krishnamurthy. Fast Video Classification via Adaptive Cascading of Deep Models. In *CVPR*, 2017.
- [30] Seung-Hwan Bae and Kuk-Jin Yoon. Robust Online Multi-Object Tracking Based on Tracklet Confidence and Online Discriminative Appearance Learning. In *CVPR*, 2014.
- [31] Min Yang and Yunde Jia. Temporal Dynamic Appearance Modeling for Online Multi-Person Tracking. *CVIU*, 2016.

- [32] Yu Xiang, Alexandre Alahi, and Silvio Savarese. Learning to Track: Online Multi-Object Tracking by Decision Making. In *ICCV*, 2015.
- [33] Alex Bewley, Vitor Guizilini, Fabio Ramos, and Ben Upcroft. Online Self-Supervised Multi-Instance Segmentation of Dynamic Objects. In *ICRA*, 2014.
- [34] Wongun Choi. Near-Online Multi-Target Tracking with Aggregated Local Flow Descriptor. In *ICCV*, 2015.
- [35] Ju Hong Yoon, Ming-Hsuan Yang, Jongwoo Lim, and Kuk-Jin Yoon. Bayesian Multi-Object Tracking Using Motion Context from Multiple Objects. In *WACV*, 2015.
- [36] Alex Bewley, Lionel Ott, Fabio Ramos, and Ben Upcroft. Alextrac: Affinity Learning by Exploring Temporal Reinforcement within Association Chains. In *ICRA*, 2016.
- [37] NVIDIA. Gst-nvinfer. https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS_plugin_gst-nvinfer.html, 2021.
- [38] Webcam Lemmer. Binnenhaven lemmer, youtube. <https://www.youtube.com/watch?v=NyzzJMwXDeo>, 2019.
- [39] See Jackson Hole. Jackson hole wyoming usa town square live cam, youtube. <https://www.youtube.com/watch?v=1EiC9bvVGnk>, 2018.
- [40] KABUKICHO. Shinjuku kabukicho, youtube. <https://www.youtube.com/watch?v=EHkMjfMw7oU>, 2020.
- [41] StarDot Technologies. Taiwan new taipei city, youtube. <https://www.youtube.com/watch?v=INR-B7FwhS8>, 2020.
- [42] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. Reducto: On-Camera Filtering for Resource-Efficient Real-Time Video Analytics. In *SIGCOMM*, 2020.
- [43] M. Kilger. A Shadow Handler in a Video-Based Real-Time Traffic Monitoring System. In *WACV*, 1992.
- [44] Kostia Robert. Video-Based Traffic Monitoring at Day and Night Vehicle Features Detection Tracking. In *ITSC*, 2009.
- [45] Tariq Abdullah, Ashiq Anjum, M. Fahim Tariq, Yusuf Baltaci, and Nikos Antonopoulos. Traffic Monitoring Using Video Analytics in Clouds. In *UCC*, 2014.
- [46] L. Snidaro, C. Micheloni, and C. Chiavedale. Video Security for Ambient Intelligence. *SMC*, 2005.
- [47] Minghu Wu, Xiang Li, Cong Liu, Min Liu, Nan Zhao, Juan Wang, Xiangkui Wan, Zheheng Rao, and Li Zhu. Robust Global Motion Estimation for Video Security Based on Improved K-Means Clustering. *JAIHC*, 2019.
- [48] Niels Haering, Péter L. Venetianer, and Alan Lipton. The Evolution of Video Surveillance: An Overview. *MVA*, 2008.
- [49] P. Chung, Yung-Ming Kuo, Chin-De Liu, and Chun-Rong Huang. Video Analysis Boosts Healthcare Efficiency and Safety. *Spie Newsroom*, 2011.
- [50] Jongse Park, Emmanuel Amaro, Divya Mahajan, Bradley Thwaites, and Hadi Esmaeilzadeh. AxGames: Towards Crowdsourcing Quality Target Determination in Approximate Computing. In *ASPLOS*, 2016.
- [51] Jiashen Cao, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. THIA: Accelerating Video Analytics using Early Inference and Fine-Grained Query Planning. *arXiv*, 2021.
- [52] Brandon Haynes, Maureen Daum, Dong He, Amrita Mazumdar, Magdalena Balazinska, Alvin Cheung, and Luis Ceze. VSS: A Storage System for Video Analytics. In *SIGMOD*, 2021.
- [53] Neil Agarwal and Ravi Netravali. Boggart: Accelerating Retrospective Video Analytics via Model-Agnostic Ingest Processing. In *arXiv*, 2021.
- [54] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. Scanner: Efficient Video Analysis at Scale. *TOG*, 2018.
- [55] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *NSDI*, 2017.
- [56] Ran Xu, Jinkyu Koo, Rakesh Kumar, Peter Bai, Subrata Mitra, Sasa Misailovic, and Saurabh Bagchi. VideoChef: Efficient Approximation for Streaming Video Processing Pipelines. In *ATC*, 2018.
- [57] Mengwei Xu, Tiantu Xu, Yunxin Liu, and Felix Xiazhu Lin. Video Analytics with Zero-streaming Cameras. In *ATC*, 2021.
- [58] Orachat Sukmarg and Kamisetty R Rao. Fast Object Detection and Segmentation in MPEG Compressed Domain. In *TENCON*, 2000.
- [59] Fatih Porikli, Faisal Bashir, and Huifang Sun. Compressed Domain Video Object Segmentation. *TCSVT*, 2009.

- [60] Fernando Bombardelli, Serhan Gül, Daniel Becker, Matthias Schmidt, and Cornelius Hellge. Efficient Object Tracking in Compressed Video Streams with Graph Cuts. In *MMSP*, 2018.
- [61] Sayed Hossein Khatoonabadi and Ivan V. Bajic. Video Object Tracking in the Compressed Domain Using Spatio-Temporal Markov Random Fields. *TIP*, 2013.
- [62] Qiankun Liu, Bin Liu, Yue Wu, Weihai Li, and Nenghai Yu. Real-Time Online Multi-Object Tracking in Compressed Domain. *IEEE Access*, 2019.
- [63] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training: Industrial Product. In *ISCA*, 2022.



SOTER: Guarding Black-box Inference for General Neural Networks at the Edge

Tianxiang Shen^{1†}, Ji Qi^{1†}, Jianyu Jiang^{1*}, Xian Wang¹, Siyuan Wen¹, Xusheng Chen¹, Shixiong Zhao¹,
Sen Wang², Li Chen², Xiapu Luo³, Fengwei Zhang⁴, Heming Cui¹

¹The University of Hong Kong ²Huawei Technologies Co., Ltd.

³The Hong Kong Polytechnic University ⁴Southern University of Science and Technology

Abstract

The prosperity of AI and edge computing has pushed more and more well-trained DNN models to be deployed on third-party edge devices to compose mission-critical applications. This necessitates protecting model confidentiality at untrusted devices, and using a co-located accelerator (e.g., GPU) to speed up model inference locally. Recently, the community has sought to improve the security with CPU trusted execution environments (TEE). However, existing solutions either run an entire model in TEE, suffering from extremely high inference latency, or take a partition-based approach to handcraft partial model via parameter obfuscation techniques to run on an untrusted GPU, achieving lower inference latency at the expense of both the integrity of partitioned computations outside TEE and accuracy of obfuscated parameters.

We propose SOTER, the first system that can achieve model confidentiality, integrity, low inference latency and high accuracy in the partition-based approach. Our key observation is that there is often an *associativity* property among many inference operators in DNN models. Therefore, SOTER automatically transforms a major fraction of associative operators into *parameter-morphed*, thus *confidentiality-preserved* operators to execute on untrusted GPU, and fully restores the execution results to accurate results with associativity in TEE. Based on these steps, SOTER further designs an *oblivious fingerprinting* technique to safely detect integrity breaches of morphed operators outside TEE to ensure correct executions of inferences. Experimental results on six prevalent models in the three most popular categories show that, even with stronger model protection, SOTER achieves comparable performance with partition-based baselines while retaining the same high accuracy as insecure inference.

1 Introduction

Driven by the remarkable success of AI [25] and edge computing [11, 17], giant companies are increasingly shifting their well-trained Deep Neural Network (DNN) models from the cloud to enormous edge devices, to compose mission-critical applications such as autopilot navigation [8, 15], home monitoring [20] and visual assistance [49]. By employing accelerators (e.g., GPU), clients of edge devices can conduct low-latency inferences without connecting with a remote server

with high latency and network instability, and clients do not have to transfer their sensitive data to the cloud.

Since obtaining accurate models requires model providers to pay substantial resources to train parameters with private datasets [10, 45], in order to preserve their competitive advantages, model providers usually require their offloaded models to keep black-box (confidential) with traditional *semantic security* guarantee [7]: a client can query the local model for results, but will learn nothing about the parameters' *plaintexts*.

To preserve model confidentiality, a pragmatic approach is to use the publicly available Trusted Execution Environments (TEE), in particular Intel SGX [14], a pervasively used CPU TEE product. SGX provides both code integrity and data confidentiality for enclaves [5, 33], and avoids severe performance downgrading caused by using traditional cryptographic tools (e.g., Homomorphic Encryption [36, 44]), making it attractive for model providers to protect their offloaded models while retaining high inference service quality.

Much prior work has explored using SGX for secure model inference, and these approaches can be summarized into two categories. First, the *TEE-shielding* approach, which runs the entire unmodified model in enclaves, achieves both model confidentiality and high accuracy same as the original model, but suffers from extremely high latency due to the limited computing resources on CPU's enclave and the lack of publicly available secure GPU devices [3, 65]. As a typical TEE-shielding work, MLCapsule [27] incurs dramatically higher latency (up to 36.1X) than insecure GPU inference for diverse workloads (§6.1).

In order to mitigate the high latency issue, the second category of work, including eNNClave [56] and AegisDNN [68], takes a *partition-based* approach to manually select a portion of sensitive model layers to run in an enclave, and partition the rest of the layers to run on an untrusted GPU for acceleration, achieving lower inference latency than TEE-shielding approaches by utilizing the strong GPU computing power.

Unfortunately, existing partition-based approaches face a fundamental *confidentiality-accuracy* dilemma: some of the approaches replace the partitioned layers' parameters with public parameters from other models composed by the same layers, which effectively protects model confidentiality but trades off the accuracy. This is because the original parameters are usually trained by model provider's private datasets tailored for a specific task [10, 45]; hence the parameters are

[†]Tianxiang Shen and Ji Qi contribute equally.

*Jianyu Jiang is the corresponding author.

exclusive to achieve the original high accuracy. In contrast, some partition-based approaches preserve the original high accuracy by holding plaintexts of partitioned operators' parameters on an untrusted GPU, which partially compromises the confidentiality by revealing a fraction of parameters' plaintexts to the adversary outside an enclave.

Our key observation to resolve this dilemma is the *associativity* property of many inference operators. *Associativity* means that the way that factors are grouped in an operator's computation does not change the final result [43]. With associativity, we can securely transform an associative operator into *parameter-morphed*, thus *confidentiality-preserved* operator by scaling its parameters with a hidden scalar in an enclave, run that morphed operator on the GPU and fully restore the execution result of that operator with the scalar hidden in an enclave. Since associative operators (e.g., *Convolution*, *Fully-connected*) widely exist in general DNN models, and these operators represent a major fraction of computation in a DNN model (e.g., 93.5% of the computation on VGG19 [61] is spent on convolution), we can achieve dramatically increased performance by partitioning morphed, computationally expensive operators to run on a GPU.

Based on this observation, we present SOTER¹, the first partition-based approach that achieves model confidentiality, low latency and high accuracy simultaneously. SOTER carries a *Morph Then Restore* (MTR) protocol for cooperative executions between kernels (enclave and GPU). Specifically, SOTER randomly selects a major fraction of associative operators, morphs their parameters with randomly generated *blinding coins* (scalars) in the enclave, and partitions these parameter-morphed operators to run on a GPU. For each client's input, SOTER executes each operator (either in an enclave or GPU) in order, transfers intermediate execution results (IR) across kernels when needed, and restores the final execution result with reciprocal coins in the enclave. A subtle case is that, under special partition cases, an adversary who observes IRs transmitted across kernels can reveal the value of coins hidden in an enclave (§4.1). Hence, SOTER additionally morphs the value of IRs before they are transmitted across kernels to hide the blinding coins, thus protecting the model confidentiality.

However, even with these steps, SOTER still faces an inherent integrity hazard in the presence of an adversary at the edge side, who can observe and manipulate any components (e.g., morphed parameters on GPU) outside the enclave to mislead the offloaded model to produce wrong output, ruining the model provider's inference service quality.

To tackle the integrity hazard, SOTER's MTR protocol generates *oblivious fingerprints* at runtime to detect integrity breaches outside the enclave in a *challenge-proof* manner [71]. Our key idea is that the associativity property can be used to efficiently generate integrity challenges at run-

System	Model	GPU	High	Inference	General
	Confidentiality	Acceleration	Accuracy	Integrity	Functionality
* MLcapsule [27]	✓	×	✓	✓	✓
* Privado [26]	✓	×	✓	✓	✓
* Occlumency [41]	✓	×	✓	✓	✓
• Serdab [21]	×	✓	✓	×	×
• Darknight [28]	×	✓	×	✓	✓
• eNNclave [56]	✓	✓	×	×	×
• AegisDNN [68]	×	✓	✓	×	✓
• SOTER	✓	✓	✓	✓	✓

Table 1: Comparison of SOTER and related systems. "*" / "•" means that the system uses either a TEE-shielding approach (*), or a partition-based approach (•).

time. Specifically, before running inferences, SOTER collects *fingerprints* of partitioned operators in enclave, which hold the ground-truth of these operators' execution results; during the inference, SOTER challenges partitioned operators with fingerprint input to ask them for execution proofs. By comparing the returned proofs with expected fingerprint output in enclave, SOTER learns whether integrity breaches occur.

To prevent an adversary from distinguishing fixed fingerprint challenges thus bypassing the detection, inspired by traditional steganography techniques [34, 52], SOTER dynamically derives new fingerprints by using existing fingerprints in enclave based on the associativity property. The new fingerprints statistically share the same distribution as client's normal input (§4.2), making fingerprints oblivious to the attacker. Therefore, by leveraging the same key observation of associativity, SOTER achieves both model confidentiality and integrity in a unified manner.

We implemented SOTER with Graphene-SGX [63], a mature framework that supports developing neural network applications with Intel SGX. We evaluated SOTER with six popular DNN models covering three popular categories, including Multi-layer Perception (MLP) [40], Convolution Neural Network (CNN) [24] and Transformer [64]. We compared SOTER to three notable TEE-based secure inference systems, covering both the TEE-shielding approach and partition-based approach. Our evaluation shows that:

- SOTER is secure. For confidentiality, SOTER effectively hid parameters' plaintexts, and achieved comparable model protection as TEE-shielding baseline under powerful model stealing attacks (§6.3); for integrity, SOTER detected any integrity breaches within ten fingerprint challenges with an overwhelmingly high probability of 99.9% (§6.4).
- SOTER is efficient. SOTER achieved up to 72.6% lower latency than the TEE-shielding baseline, and had moderate latency (up to 1.2X) compared to partition-based baselines, while the baselines do not have integrity protections (§6.1).
- SOTER is accurate. SOTER retained the same high accuracy as insecure inference, while the most efficient partition-based baseline caused 1.1%~5.5% accuracy drops.

Our major contribution is the MTR protocol, the first work that achieves model confidentiality, low latency and high ac-

¹SOTER is an ancient Greek god, guards safety against cosmic chaos.

curacy with integrity protection for secure model inference. Compared to existing relevant baselines, SOTER achieved comparable strong confidentiality as the TEE-shielding approach, comparable low-latency as the partition-based approach, high accuracy same as insecure inference, and overwhelming high probability of detecting integrity breaches outside an enclave. This makes SOTER unique to greatly promote the prosperity of AI on edge devices, encouraging enormous model providers to develop powerful models and deploy them on third-party edge devices. Also, our MTR protocol can generally protect model inference on the cloud when the model provider does not trust the owner of cloud servers that host the model. SOTER’s source code is released on github.com/hku-systems/SOTER.

In the rest of this paper: §2 introduces background; §3 gives an overview of SOTER; §4 describes SOTER’s design; §5 covers SOTER’s implementation; §6 shows our evaluation; §7 discusses related work and §8 concludes.

2 Background

2.1 Deep Neural Network

A DNN model (in short, model) can be represented as a sequence of connected *layers* with each layer assigned a set of *operators*, as shown in Figure 1. An operator is either a *linear operator* or a *nonlinear operator* where a linear operator is weighted by *parameter matrices* (in short, parameters).

Inference workflow. Figure 1 shows the inference workflow. A model M passes an input X (e.g., an image) through layers of operators to compute *logits* [30], normalizes logits with *softmax* function to produce a probability vector, and assigns a class with the highest probability to input X as the class label. Without losing generality, we use image classification as an example to illustrate model composition in the following discussions.

Associativity of DNN operators. Operators are the basic building blocks of a DNN model, among which linear operators have been proven to take up the majority of computation resources in general model inferences [1, 70].

Many DNN models deployed at the edge are built on top of *associative* operators. Suppose we have an input X and a scalar μ , a DNN operator F is associative if

$$(\mu^{-1} * \mu) \cdot F(X) = \mu^{-1} \cdot F(\mu * X) \quad (1)$$

Linear operators, including computationally expensive *convolution* and *fully-connected*, have the associativity property as they conduct linear transformation on input data. For instance, take the convolution operator as an example: as shown in Figure 1, if we multiply each element in the convolution kernel by 2^{-1} (i.e., $\mu = 2^{-1}$), we will get the output $2^{-1}R$, while it always holds $(2 * 2^{-1}) * R = 2 * (2^{-1}R)$ in Equation 1. This property applies to other linear operators as well.

For nonlinear operators that conduct nonlinear transformation on data, most of them do not have an associativity property (e.g., *Sigmoid*). However, interestingly, under specific

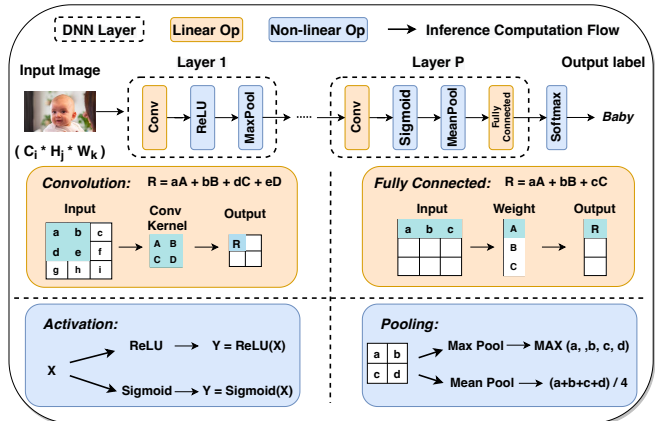


Figure 1: An overview of the DNN model.

constraints, some nonlinear operators can also be associative. Take the most commonly used *ReLU* as an example: the *ReLU* function, $F(x) = \text{Max}\{0, X\}$, is *scale-invariant* when $\mu > 0$, i.e., $F(\mu x) = \text{Max}\{0, \mu X\} = \mu F(X)$. Hence, given a scalar μ , Equation 1 applies to *ReLU*. Similar to *ReLU*, the *pooling* function is associative as well.

Operators that satisfy Equation 1 also meet a variant of the associativity property. Given that

$$F(X_1) = y_1, F(X_2) = y_2, \dots, F(X_n) = y_n$$

it always holds

$$F\left(\sum_{i=1}^n \mu_i * X_i\right) = \sum_{i=1}^n \mu_i * y_i \quad (2)$$

SOTER leverages Equation 1 as the key weapon to produce parameter-morphed, thus confidentiality-preserved operators to run on GPU for acceleration, and restore accurate results in enclave. SOTER uses Equation 2 to efficiently generate oblivious fingerprints for integrity checking at runtime. We illustrate our detailed designs in §4.

2.2 Intel SGX and Related Work

Intel Software Guard eXtension (SGX) [14] is a pervasively used hardware feature on commodity CPUs. SGX provides a secure execution environment called enclave, where data and code execution cannot be seen or tampered with from outside.

As shown in Table 1, there are two categories of SGX-based work that provides secure inference service. The first category is the TEE-shielding approach that runs all inference computations within enclave (e.g., MLCapsule [27], Privado [26], Occlumency [41]). Such an approach shields the entire model in enclave to hide parameters’ plaintexts, but fails to achieve low inference latency owing to the computational bottleneck of CPU and the lack of publicly available trusted GPU [3, 65].

The second category is the partition-based approach that runs a portion of model layers in enclave and runs the rest of the layers on a GPU for acceleration. AegisDNN [68] only shields partial critical model layers in enclave and accelerates other plaintext layers on GPU. To decide which layers should

be partitioned, AegisDNN takes a user-specified argument (i.e., deadline for an inference task), uses silent data corruption mechanism [23] to learn each layer’s criticality, and partitions uncritical (plaintext) layers to GPU to meet the stringent deadline. eNNclave [56] argues that the feature extraction operators (e.g., convolution) of different models are generally transferable, hence it replaces partitioned operators’ parameters with pre-trained parameters of other models, which sacrifices inference accuracy. This is because, model providers usually train their models with private datasets tailored for a specific task, thus each parameter in the trained model is exclusive to achieve high accuracy [10, 45, 72]. Also, such an approach is only applicable to specific models where public parameters are available. Serdab [21] and Darknight [28] assume the model is deployed on a trusted cloud. Instead of protecting model confidentiality, they have an orthogonal goal of protecting users’ data privacy on the cloud.

3 Overview

3.1 System setup

We consider a client-side inference scenario shown in Figure 2. Different from a cloud-side inference scenario (e.g., Delphi [44], Gazelle [36]) where the model is hosted on the trusted cloud server, we consider the model provider offloads its model to the client’s untrusted edge device to run model inferences. The device constantly takes sensitive queries from the client and sends inference results back to the client.

3.2 Security model

We consider an honest model provider that provides the correct model requested by the client, and the model is offloaded to run on an SGX-equipped third-party edge device. We trust the hardware and firmware of Intel SGX, which ensure that code and data in enclave can not be seen or tampered with from outside. However, any components outside the enclave are untrusted.

We consider a malicious edge-side attacker outside the enclave that aims to (1) steal the parameters of the offloaded model, and (2) perturb any components outside the SGX to modify the inference results. An edge-side attacker could be a business competitor who wants to steal the model for competitive advantages and ruin the inference service to screw up the model provider’s reputation [12, 13, 54]. Even worse, the integrity attack against edge-side model inference could pose severe threats to edge users. For instance, an attacker may hack into a self-driving system running with an obstacle detection model, and perturb the model parameters to produce incorrect navigation instructions to the car [9].

Semantic security. Similar to prior secure inference systems, namely the MLcapsule [27] and eNNclave [56], SOTER aims to achieve model confidentiality with the semantic security guarantee: knowing only ciphertexts, it must be infeasible for a computationally-bounded adversary to derive significant information about the plaintexts [7]. In the secure infer-

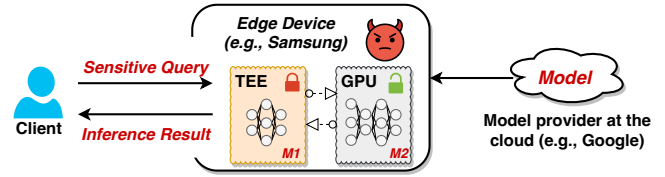


Figure 2: The client-side inference scenario.

ence scenario, it captures the requirement that the parameters’ plaintexts cannot be derived from any data observed by the adversary.

Note that, we do not hide information that is revealed by the results of inference queries, and we focus just on protecting the parameters’ plaintexts. Protecting against attacks that exploit the leakage of inference results is a complementary problem to that solved by SOTER. We give a detailed illustration of these attacks and potential mitigations in §7.

3.3 System overview

SOTER’s two-phase design. SOTER’s protocol consists of an offline *preprocessing phase*, and an online *inference phase*. Specifically, the preprocessing phase is independent of the client’s query input which changes regularly. We assume the offloaded model from the server is static, if the model changes, then both parties should re-run SOTER’s preprocessing phase. After preprocessing, during the inference phase, the client sends query input to get the eventual result. Note that, SOTER is best suited for applications whose inference is latency-sensitive, but is usually not performed frequently enough to take up all computational resources needed for preprocessing. **SOTER’s workflow.** In Figure 3, we show how SOTER leverages the general associativity property of DNN operators to automatically partition a DNN model.

In the preprocessing phase: during **P1~P3**, a SOTER client conducts standard SGX attestation to the server for obtaining the model M and decryption keys KEY_M , and then loads the encrypted model M in a layer-wise manner by decrypting with KEY_M locally. In **P4** and **P5**, SOTER extracts the model architecture, statically filters out all *associative* operators that meet Equation 1, and then invokes SOTER’s MTR protocol for model partitioning.

Specifically, in **P5**, SOTER’s MTR protocol runs as follows: With a given partition ratio θ , SOTER randomly selects $\theta\%$ associative operators, and generates (1) *fingerprints* of the selected operators for integrity checking (used in the inference phase), and (2) random scalars for all operators. These scalars serve as the *blinding coins* to hide the parameters [44] and are always kept secret in enclave. With the associativity property (Equation 1), SOTER morphs every selected associative operator by multiplying each element in the operator’s parameter matrices with a blinding coin (the computation result can be restored by using the reciprocal coin), and then partitions the selected operators with morphed parameters to GPU. In Figure 3, the selected operators (in green) are morphed with

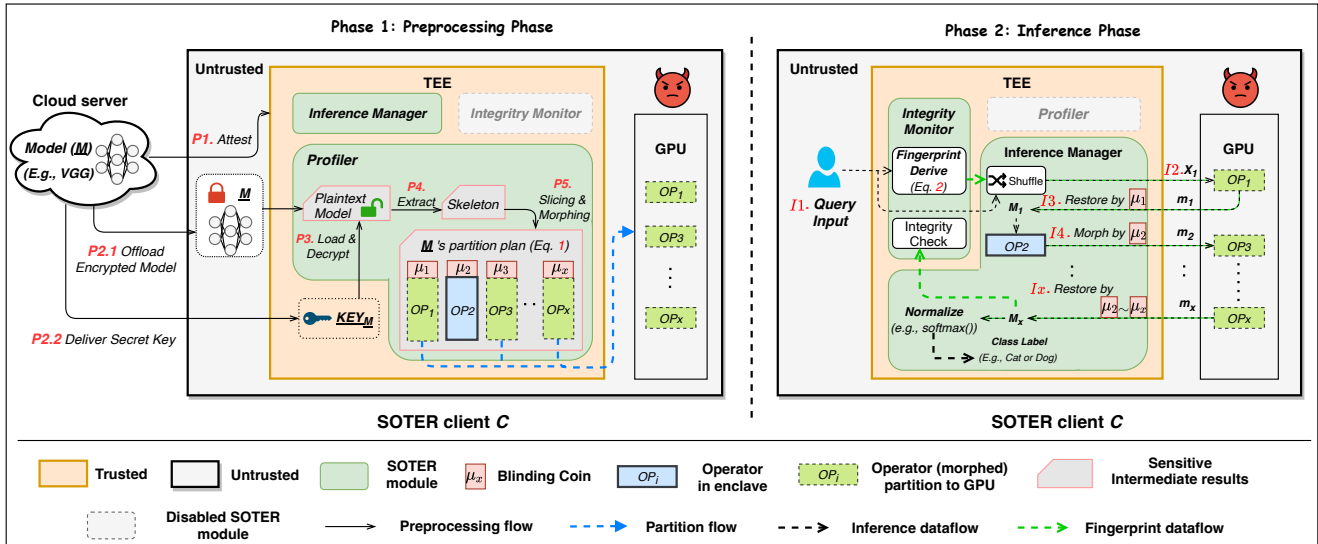


Figure 3: The architecture of SOTER in a modular manner. SOTER’s secure inference protocol has three modules (shielded in green) running in an enclave. An adversary can observe and manipulate any components outside the enclave.

corresponding coins, e.g., the parameters of the third operator OP_3 are morphed with its coin μ_3 and partitioned to GPU, while the second operator (in blue) is kept within an enclave.

Note that, we choose to run the preprocessing procedures (P1~P5) at the client-side because client C might select his θ according to his GPU capability, e.g., choose a smaller θ if the GPU has limited usable memory.

In the inference phase, there are two types of data flows: the inference flow (black dotted line) and the fingerprint data flow (green dotted line). The life-cycle for the inference data flow is: (I1) The client C sends a query input (i.e., data for inference) to the trusted inference manager module and shuffles with the fingerprint input (introduced below). (I2) The inference manager forwards C ’s input X_1 to the GPU, and the first *morphed* operator OP_1 takes X_1 as input and computes the output m_1 . (I3) Since the next operator OP_2 is expected to run within enclave according to the plan made in the preprocessing phase, the inference manager takes the computation result of previous partitioned operators (i.e., m_1) as the input, restores m_1 with all previous partitioned operators’ blinding coins since the last kernel switch (in Figure 3’s circumstance, SOTER restores with only OP_1 ’s blinding coin μ_1), and gets the restored result M_1 . (I4) After that, the second operator OP_2 computes with M_1 and gets an intermediate result (IR), then morphs IR with the OP_2 ’s blinding coin μ_2 and forwards the result m_2 to the partitioned operator OP_3 in GPU.

The above procedures (I2~I4) repeat until all partitioned operators in GPU are computed. In (Ix), with the last computation result m_x from OP_x , SOTER restores the real result with all partitioned operators’ coins since the last kernel switch (same as I3). In this case, SOTER restores with $\mu_2 \sim \mu_x$ and gets the real inference result M_x . Last, SOTER runs a standard normalization to get the class label for client C .

The fingerprint data flow is used to detect integrity breaches of partitioned operators outside an enclave. Specifically, with a set of fingerprints produced in the preprocessing phase (P5), SOTER dynamically derives new fingerprints by utilizing the general associativity of Equation 2, injects these fingerprints into the inference data flow to check whether the partitioned parameters have been maliciously modified by an adversary outside the enclave.

SOTER’s generality. SOTER supports general neural networks. SOTER’s key insight into partitioning neural networks lies in the broad associativity of common inference operators (§2.1), including all linear operators (e.g., *convolution* and *fully-connected*) and typical nonlinear operators (e.g., *ReLU*, *Max-pool*, and *Mean-pool*). Hence, SOTER theoretically supports all neural networks (including recurrent neural networks [69]), but exhibits varying performance gains (compared with running the entire neural network in TEE) depending on the ratio of associative operators in neural networks (§6.1). SOTER also supports general TEEs (e.g., ARM TrustZone [53] and AMD SEV [58]) as long as the TEE provides data confidentiality and code integrity guarantees that SOTER’s MTR protocol requires.

4 Protocol Description

This section describes SOTER’s MTR protocol. At a high level, MTR utilizes the general associativity property of DNN operators to automatically profile a model (with Equation 1), randomly selects a portion of associative operators, and then morphs these operators’ parameters with hidden *blinding coins* in enclave to hide parameters’ plaintexts (§4.1). Besides, to tackle the integrity threat (described in §3.2), stemming from the same observation of the general associativity property, MTR dynamically derives *oblivious fingerprints* (with

Table 2: SOTER’s protocol variables.

Variable	Description
$\Theta(\text{default} = 0.8)$	Portion of associative operators partitioned to GPU.
$O_g \mid O_e$	Partitioned operators in GPU / maintained in TEE.
$O(O_g \cup O_e)$	The whole set of model M ’s operators.
$Para(O_i)$	Parameters of operator O_i .
FP_{O_i}	Fingerprint of operator O_i .
μ_i	Blinding coin of operator O_i .

Equation 2) and uses fingerprints to check the integrity of partitioned operators in a *challenge-proof* manner (§4.2). Table 2 shows the variables used in the MTR protocol.

4.1 Morph Then Restore (MTR) protocol

The MTR protocol is divided into two stages: a *morphing* stage and a *restore* stage. In the morphing stage, SOTER first makes *blinding coins* for every operator (including both associative and non-associative operators), and then makes a partition plan with a given partition ratio θ . Then, in the restore stage, SOTER runs inference across GPU and enclave, and restores inference results in enclave when needed. Algorithm 1 and 2 show the MTR protocol.

Stage 1-1: Morphing with blinding coins. SOTER assigns every operator with a randomly generated scalar, which serves as the blinding coin to hide the plaintext value of parameters. Given an operator O_i (the partition plan is described in **Stage 1-2**), SOTER morphs $Para(O_i)$ by multiplying each element in $Para(O_i)$ with the corresponding coin μ_i . Note that, SOTER requires periodically updating of the coins to avoid potential chosen plaintext attacks [6, 37], as the morphing is completely linear. According to the hill cipher theory [48], each coin can tolerate up to n^2 attacks (i.e., n^2 inferences) where n is the parameter matrix’s size of the operator that the coin applies to. Hence, SOTER updates a morphed operator with matrix size n every n^2 inferences. Notably, this updating can be done off the critical path when the inference tasks are not busy (§5).

- **Protect coins during kernel switches.** One subtle case is that, SOTER assigns random coins to every operator (rather than selected operators for partitioning) to avoid potential information leakage during kernel switches, i.e., when intermediate results (IRs) are transmitted between the enclave and GPU. We illustrate our design by giving a running example.

- **Running example.** We demonstrate our idea with the example in Figure 4, which is a common architecture in typical DNN models. SOTER selects a portion of model operators to run in the TEE enclave (green portion), and partitions the remaining operators to run on the untrusted GPU (red portion). An adversary outside the enclave is attempting to deduce the blinding coins used to morph the GPU operators according to the attacker-visible IRs (Y_1 and Y_2).

We begin with the **Before** case, in which only the partitioned GPU operators are protected by SOTER’s blinding coins. OP_1 (on the GPU) is a linear operator and OP_2 (in

Algorithm 1: MTR protocol at client c (offline) (§4.1).

```

1 Function partition( $O$ ) do
2   foreach operator  $op$  in  $O$  do
3      $op.partition \leftarrow \text{False}; op.index \leftarrow O.index(op);$ 
4     if  $op.index > W$  then
5       if  $0 < \text{normalize}(\text{sgx\_read\_rand}()) < \Theta$  then
6          $op.partition \leftarrow \text{True};$ 
7       if  $op$  is associative &  $op.partition$  then
8         foreach element  $e$  in  $para(op)$  do
9            $e \leftarrow \mu_{index} \times e;$  // Morph
10           $O_g.add(op);$  // GPU operators
11        else
12           $O_e.add(op);$  // TEE operators
13       $O_g.copyTo("cuda");$  // Partition to GPU

```

enclave) is a non-linear *ReLU* operator. If we only morph OP_1 with μ_1 and do not assign a coin to OP_2 , then, with a client input X , OP_1 outputs $y_1 = X(\mu_1 * OP_1)$ and OP_2 outputs $y_2 = ReLU(y_1/\mu_1) = ReLU(X * OP_1)$. However, since the *ReLU* operator only filters out negative values in parameter matrices and does not transform a scalar, an adversary who observes both y_1 and y_2 will directly infer the value of coin μ_1 , violating the confidentiality of the partitioned OP_1 (concretely, $Para(O_i)$). To tackle this problem, we assign coins to all operators rather than partitioned GPU operators only. As demonstrated in the **After** case of Figure 4, by assigning coin μ_2 to OP_2 , OP_2 will output $y_2 = \mu_2 * ReLU(X * OP_1)$. An adversary observes $\mu_1 * \mu_2$ but has no way of inferring either μ_1 or μ_2 .

Overall, by morphing both the enclave and the partitioned GPU operators with blinding coins stored in TEE enclave, an attacker cannot deduce the blinding coins from the IRs, ensuring the confidentiality of the model parameter’s plaintexts. **Stage 1-2: Partitioning model with hidden operators.** With all coins prepared, given a partition ratio θ , we automatically partition a portion of associative operators (that are morphed) to GPU for inference acceleration. The *partition* function in line 1~13 of Algorithm 1 shows the pseudo-code.

Specifically, in the preprocessing phase, SOTER iterates every operator in model M and randomly selects a portion of associative operators and adds them to the partition set O_g . For instance, with $\theta = 0.8$, an associative operator would have an 80% chance to be partitioned to GPU. After all operators are iterated, all operators in O_g are partitioned to GPU for inference acceleration.

Note that, SOTER always keeps top- W ($W=2$ by default) operators in enclave even if these operators are associative (Line 4). This design choice is made to ensure the input to the first several partitioned operators on GPU (e.g., X_1 in Figure 3) are always unknown to the adversary, such that we can stealthily check the integrity of all partitioned operators by injecting fingerprint challenges (more details in §4.2).

Stage 2: Guarded black-box inference. Next, we present the

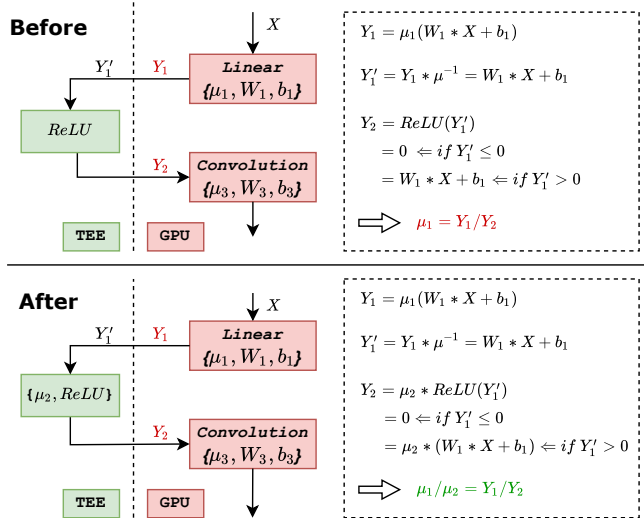


Figure 4: Morphing only partitioned operators' parameters can leak the blinding coins in some partition cases.

end-to-end black-box inference process given that we have prepared coins and partitioned a portion of operators to GPU.

As shown in Algorithm 2, SOTER first computes the reserved top- W operators in enclave, and then iterate **I2**~**I4** as follows. **I2**: SOTER finds the longest length of consecutively partitioned operators in O_g and computes the inference result by forwarding the input through these operators (line 6~11). **I3**: Then SOTER copies the result back to enclave, restores the real inference result by multiplying the reciprocal coins of all operators in the last step (line 13), and then forwards the result to operators maintained in enclave (before the next copy to GPU). **I4**: SOTER morphs the forwarding result with all coins of enclave operators in the last step (line 18), and then copies back to GPU. This procedure terminates when all partitioned operators on GPU are iterated. Then SOTER normalizes the final result and returns the result to the client.

4.2 Integrity checking with fingerprints

Obliviousness requirement. To detect integrity breaches outside an enclave (as described in §3.2), a straw man approach could be using a fingerprint (with ground-truth input/output pair) to *challenge* the partitioned operator on GPU to recompute the fingerprint input to provide *proof*, and report a crime if the proof is different from the expected output.

However, such an approach needs to be oblivious. Since the adversary can continuously monitor the inference process and observe every intermediate result (IR) transmitted from enclave to GPU, if using only a fixed set of fingerprints, the adversary can easily distinguish those fixed challenges among all IRs, and bypass our detection by returning correct proofs. **The timeliness-obliviousness dilemma.** Unfortunately, trivially using different fingerprints for integrity checking brings a *timeliness-obliviousness* dilemma.

Specifically, generating new fingerprint input and pre-

Algorithm 2: MTR protocol at client c (online) (§4.1).

```

1 Function secure_inference() do
2    $\nabla$  I1.
3    $X \leftarrow \text{copyTo}(\text{"enclave"})$ ;
4   foreach index  $i \in \text{normalize}(W)$  do
5      $X \leftarrow O_i(X)$ ; // Top-W Inference
6   start  $\leftarrow$  next op in  $O_g$ ; end  $\leftarrow$  next op in  $O_e$ ;
7   while start < | $O$ | do
8      $\nabla$  I2.
9      $X \leftarrow \text{copyTo}(\text{"cuda"})$ ;
10    foreach index  $i \in \text{range}(\text{start}, \text{end})$  do
11       $X \leftarrow O_i(X)$ ; // GPU Inference
12     $\nabla$  I3.
13     $X \leftarrow \prod_{\text{start}}^{\text{end}-1} \mu_i^{-1} \times X$ ; // Restore
14    start  $\leftarrow$  end;
15    end  $\leftarrow$  start from  $O_{\text{end}}$  find the next op in  $O_g$ ;
16    foreach index  $i \in \text{range}(\text{start}, \text{end}-1)$  do
17       $X \leftarrow O_i(X)$ ; // TEE Inference
18     $\nabla$  I4.
19     $X \leftarrow \prod_{\text{start}}^{\text{end}-1} \mu_i \times X$ ; // Morph
20   $X \leftarrow \text{normalize}(X)$ ;
21  return Top $_1(X)$ ; // Final result

```

computing expected output in CPU would cause late detection, because CPU computation for an operator is extremely slow (up to 30x slower inference speed for linear operation); on the other hand, if we use a fixed set of fingerprints for challenging the integrity, we could timely detect breaches. However, the detection would be easily bypassed by the adversary because all fingerprints are now distinguishable.

SOTER's insight and approach. Stemming from the same observation of parameter morphing (§2.1), we tackle this challenge by efficiently generating new fingerprints in CPU without re-computing operators. At a high level, we take two steps by first making a set of *cornerstone fingerprints* for each partitioned operator, and then encapsulating new fingerprints from these cornerstone fingerprints with a constant cost at runtime based on the associativity property.

Step 1: Preparing cornerstone fingerprints. During the pre-processing phase, SOTER prepares K (by default $K=10$) cornerstone fingerprints for each partitioned operator. A fingerprint is a two-element tuple: $\{\text{input}, \text{output}\}$, where the input is a randomly generated matrix and the output is pre-computed by forwarding the input through a ready-to-partition operator. The whole preparing procedure is running within enclave, such that the correctness of each cornerstone fingerprint can be guaranteed.

Step 2: Efficiently deriving new fingerprints in TEE. Now with K cornerstone fingerprints for each partitioned operator, SOTER efficiently derives oblivious fingerprints at runtime by leveraging the general associativity property. Specifically, with a fixed set of cornerstone fingerprints $FP_O[K]$ for each

partitioned operator O , we randomly select T fingerprints and generate T random coefficients with SGX’s trustworthy random number generator. Then, by applying the associativity property (Equation 2), we generate a new fingerprint FP_{New} as follows: the input for FP_{New} is $\sum_{i=1}^T T_i * FP_O[i].input$, i.e., we multiply each selected cornerstone fingerprint’s input with a corresponding coefficient and add them all as the new input; For the output, since the associativity property implies that the corresponding output for a group of associated operators has the same transformation as the input, thus we can directly get the expected output as $\sum_{i=1}^T T_i * FP_O[i].output$ without conducting slow CPU inference for a given new input.

Step 3: Challenge-proof fingerprint issuing. For fast detection of integrity breaches, rather than submitting fingerprint challenges for random user inference tasks, SOTER submits fingerprint challenges for every user inference task. By doing so, SOTER can detect any integrity breaches within only ten fingerprints with an overwhelmingly high probability of 99.9% according to our theoretical analysis and evaluation results (§6.4).

In detail, SOTER issues a *challenge* whenever a kernel switch (an IR transmitted from enclave to GPU) happens. When an IR (produced by client’s query) is sent to a partitioned operator O , we challenge the integrity of O by sending the IR to O together with a new $FP_O.input$ (produced in Step 2), and compare the returned *proof* with the expected $FP_O.output$. Any mismatch reveals an integrity breach of partitioned operator O .

Whenever integrity breaches are detected, SOTER can simply abort the inference process and restart the preprocessing phase to prepare new blinding coins and re-morph the GPU operators with the procedures described in §3.3.

5 Implementation

We built SOTER on top of PyTorch C++ [51] with 5.3K LoC. We chose PyTorch C++ as it incurs less memory footprint compared to PyTorch, and previous work [63] has shown that running programs with a high memory footprint within SGX could incur prohibited overhead due to SGX’s small memory capacity. SOTER uses Graphene-SGX to run SOTER’s components within an SGX enclave (denoted as SOTER-Graphene), because Graphene-SGX provides a library OS for running applications within the enclave without any modifications.

Due to SGX’s limitation (§2.2), Graphene-SGX cannot directly access GPU within the enclave. SOTER tackles this problem by spawning an extra PyTorch process (SOTER-GPU) outside the enclave for offloading computations to GPU. SOTER-Graphene and SOTER-GPU communicate through shared memory (untrusted and not protected by SGX). We did not choose to modify Graphene-SGX to forward GPU computations at the syscall-level (e.g., by modifying `ioctl`), because doing so results in frequent enclave transitions for each GPU task. This is because launching one GPU task requires multiple system calls (e.g., `ioctl` and `mmap`).

Runtime Construction. SOTER builds the runtime according to the two-phase design (§3). In the *preprocessing* phase, SOTER bootstraps the enclave with the standard SGX attestation library [14], and decrypts the model (in ONNX format [42]) sent from the server in the client enclave. The enclave bootstrapping takes 1.84~2.92 seconds. Then, the decrypted model is processed by `enclave_model_dispatcher`, which randomly selects a major fraction of associative operators and morphs these operators’ parameters with random scalars produced by the SGX trustworthy source (`sgx_read_rand`).

In the *inference* phase, SOTER-Graphene runs inferences on DNN layers stored within TEE. When an inference computation needs to be offloaded outside the enclave, SOTER-Graphene serializes the activation using `pickle`, pushes the data to the shared memory and hands over to SOTER-GPU; SOTER-GPU deserializes the data and the morphed model, computes the results, and pushes the result to the shared memory; SOTER-Graphene retrieves the results and continues execution.

Optimization. First, SOTER reduces the memory footprint by reusing a single `paraheap` buffer to store operators’ parameters, and gradually loading and decrypting parameters from disk when they are required for computations in CPU. Second, SOTER enables SGX paging [14] to support the execution of large DNN models. Note that all baseline systems we evaluated were also enabled with these two optimizations. Third, SOTER takes unused CPU cycles to provision new coins and produce newly morphed operators in TEE (§4.1), to replace staled partitioned operators on GPU in a nonstop manner.

6 Evaluation

Testbed. Our evaluation was conducted on a server with 2.60GHZ Intel E3-1280 V6 CPU, 64GB memory, and SGX support. The partitioned computations were performed on a co-located Nvidia 2080TI GPU with 11 GB physical memory.

Baseline systems. To evaluate the performance of SOTER, we compared SOTER with three notable TEE-based secure inference systems, namely MLCapsule [27], AegisDNN [68] and eNNclave [56]. MLCapsule is a popular TEE-shielding open-source project, which shields the entire model within the CPU enclave without any modification to parameters, thus achieving model confidentiality, integrity (no partitioned operators outside enclave) and retaining high accuracy as the original trained model.

AegisDNN and eNNclave are two state-of-the-art partition-based systems that empower accelerator-assisted low-latency inference same as SOTER (Table 1). AegisDNN only shields partial critical model layers in enclave and accelerates other plaintext layers on a GPU. To decide which layers should be partitioned, AegisDNN takes a user-specified argument (i.e., deadline for an inference task) and uses a silent data corruption mechanism [23] to learn each layer’s criticality. AegisDNN is not open-source so we implemented all its protocols. eNNclave handcrafts a partition plan by manually replacing

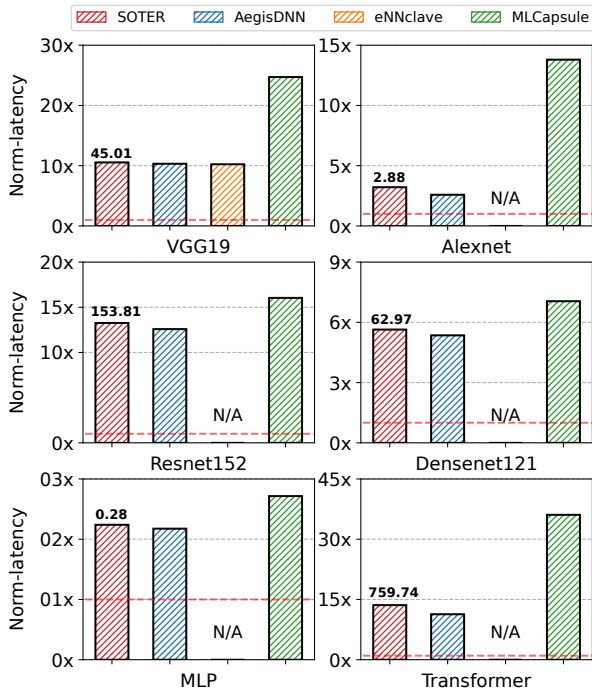


Figure 5: Normalized latency of all systems running six prevalent models. All systems’ latency results are normalized to **insecure GPU** inference latency (red dotted line). The value on each red bar indicates SOTER’s averaged inference latency.

some layers’ parameters with public insensitive parameters, and running these parameter-precision downgraded layers on a GPU. Hence, eNNclave achieves model confidentiality at the expense of accuracy, and is only suitable for specific models with publicly available parameters of known layers. We ran eNNclave based on its open-source implementation.

Note that, both AegisDNN and eNNclave do not provide integrity protection for a partitioned model. While some approaches (e.g., Darknight [28]) have integrity protection for computations on untrusted accelerators, they are not designed to protect model confidentiality at the untrusted edge; hence they are orthogonal to the goals of SOTER.

Models and datasets. We evaluated all baseline systems with six well-studied DNN models in the three most popular categories that are widely used in the deep learning community, including a Multi-layer Perception (MLP) model, four Convolution Neural Network models including Alexnet (AN) [38], Densenet121 (DN) [31], VGG19 (VGG) [61], Resnet152 (RN) [29], and a Transformer (TF) [64]. We used the open-source release of each model.

We conducted our study on two representative datasets, ImageNet [16], and WNMT [57], targeting both computer vision (CV) and natural language processing (NLP) tasks. ImageNet is a full-scale image dataset that contains 600k images from 1k classes for CV studies; WNMT is the major translation dataset that has been used in recent NLP studies.

Default setting. By default, we ran all experiments by sequen-

SOTER’s inference results (in milliseconds)

Model	MLP	AN	VGG	RN	DN	TF
P1: CPU (TEE)	0.19	1.65	25.38	92.18	41.65	439.52
P2: GPU	0.05	0.71	14.24	33.97	13.71	204.93
P3: Kernel Switch	0.01	0.18	0.83	25.98	5.6	41.52
P4: Integrity Check	0.03	0.34	4.56	14.75	6.02	73.77
End-to-end (P1+P2+P3+P4)	0.28	2.88	45.01	153.88	62.97	759.74

Table 3: End-to-end latency breakdown of SOTER.

AegisDNN’s inference results (in milliseconds)

Model	MLP	AN	VGG	RN	DN	TF
P1: CPU (TEE)	0.19	1.54	22.89	88.14	36.75	404.87
P2: GPU	0.07	0.67	19.96	52.3	20.07	198.64
P3: Kernel Switch	0.01	0.12	0.85	7.12	1.81	29.61
End-to-end (P1+P2+P3)	0.27	2.33	43.7	146.56	58.63	633.12

Table 4: End-to-end latency breakdown of AegisDNN.

tially feeding the inference data (i.e., *input_batch_size=1*). Unless conducting sensitivity studies, we ran SOTER with 80% probability of partitioning an associative operator to run on a GPU (i.e., *selective partition ratio=0.8* in §4.1).

Our evaluation focuses on the following questions:

§6.1 How efficient is SOTER compared to baselines?

§6.2 How sensitive is SOTER’s performance to different partition ratio?

§6.3 What could be leaked with and without SOTER?

§6.4 How robust is SOTER under integrity breaches?

6.1 End-to-end inference performance

We first investigated the end-to-end inference latency of SOTER and three baseline systems with six prevalent models. All reported measurements are the averaged inference latency of 10k independent inference input.

Figure 5 shows the comparison of normalized inference latency, where all systems are normalized to the insecure GPU inference, which was measured by directly running model inference on GPU without protection. SOTER’s end-to-end latency (in milliseconds) is reported on its bar. N/A means a system cannot handle such a case because eNNclave only reports its method on partitioning VGG-series models. Overall, SOTER’s latency was $1.21X \sim 4.29X$ lower than MLCapsule when running inferences on the same model. This is because MLCapsule shields the entire model in the CPU enclave without involving GPU, and CPU is at least one order of magnitude less efficient than GPU in executing inference tasks [62].

Meanwhile, SOTER incurred $1.03X \sim 1.27X$ higher latency compared to AegisDNN for two reasons. First, SOTER additionally enforces integrity protection of partitioned operators on GPU (§4.2) while AegisDNN does not support integrity check. Second, AegisDNN partitions models in a coarse-grained manner: by packing consecutive layers into blocks and partitioning these blocks to run on a GPU, AegisDNN reduces the number of kernel switches (memory copy) between enclave and GPU. In contrast, SOTER partitions model in a fine-grained operator granularity, thus may incur more kernel switches on some models. Next, we evaluate this moderate performance downgrading with a detailed latency breakdown.

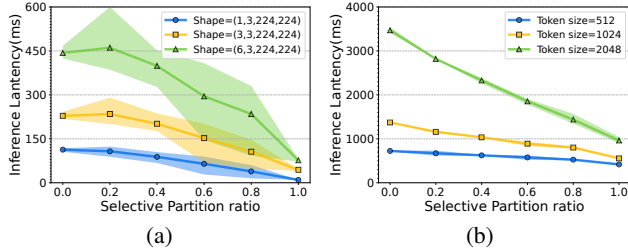


Figure 6: Running SOTER on (a) VGG19 with different input shapes (*batch_size, channels, height, weight*), and (b) Transformer with different token size.

Latency breakdown. To understand SOTER’s low latency and moderate overhead, we recorded the time taken for each step of the workflows in SOTER and AegisDNN, as shown in Table 3 and Table 4.

In addition to the integrity check and kernel switch overhead, we observed that SOTER took more time for enclave computations by comparing **P1** in both tables. This is because SOTER provides confidentiality for all model operators by either shielding plaintext operators in enclave, or selectively morphing associative operators to run on a GPU and reverting concealed execution results to accurate results in enclave with preserved blinding coins (§4.1). In contrast, AegisDNN only protects confidentiality for partial operators in enclave, and leaves other operators’ plaintext parameters to the untrusted GPU. Therefore, SOTER paid additional overhead for parameter morphing and restoration in enclave.

Besides, when running models with complex dependencies among operators (Resnet, Transformer), SOTER’s operator-level partition protocol incurred more frequent kernel switches than AegisDNN (**P3**), but such design protects the entire model architecture from being leaked. As recent work on AI demonstrates that model architecture has a significant impact on achieving high inference accuracy [2, 22], a complex, thus potentially well-designed architecture is urgent to get protected. In contrast, AegisDNN partitions a bunch of consecutive layers to GPU thus leaking architecture information to the adversary outside enclave. We will further investigate the information leakage in all systems in §6.3.

Overall, compared to the TEE-shielding baseline (MLCapsule [27]), SOTER achieves lower inference latency by securely partitioning most associative operators to run on an untrusted GPU, and the performance gain brought by GPU computations on associative operators dominates the overhead for TEE paging and TEE-GPU interaction (Table 3). Typical associative operators (e.g., convolution and fully connected) are proven to be the major computational bottleneck in neural networks. For example, 93.5% of the computation on the classic VGG19 model is spent on convolution [61]. However, compared with the partition-based baselines (eNnclave [56] and AegisDNN [68]), SOTER incurs slightly higher (up to 1.27X) latency for using fingerprints to detect integrity breaches (latency breakdown in Table 3 and Table 4).

No accuracy loss. SOTER retained high inference accuracy provided by original models, because SOTER ran either unmodified operators in enclave or morphed associative operators in GPU, while concealed execution results of morphed operators can be reverted with the associativity property (§2.1). MLcapsule directly shields the entire model in enclave without modification so it retained high original accuracy; AegisDNN partitions plaintext model layers to run on a GPU so it incurred no accuracy downgradation as well; eNnclave obfuscates partitioned operators’ parameters by replacing them with public parameters. Since public parameters are not tailored for a given task, eNnclave incurred 1% ~ 5.5% accuracy drops on the evaluated VGG19 model.

In summary, SOTER achieves much lower inference latency than MLcapsule, but SOTER incurs slightly higher latency than AegisDNN and eNnclave because SOTER pays additional effort to provide stronger model confidentiality and integrity, and retains high accuracy same as original models.

6.2 Sensitivity

We tested different partition ratios (denoted as θ) to look into the performance sensitivity of SOTER. The experiments were conducted on VGG19 with different input shapes, and we ran different values of θ for each input shape; each experiment was conducted ten times, as shown in Figure 6a.

The shading next to each line is the performance jitter caused by different computation volumes of randomly selected associative operators partitioned to GPU. With an increased θ , SOTER’s inference latency dropped accordingly because more associative operators were selected and partitioned to a co-located GPU for acceleration. Interestingly, we observed that when θ is small (0.1 ~ 0.2), the latency of $\theta = 0.2$ could be even larger than $\theta = 0.1$. This instability is because, with a small partition ratio, the partitioned operators were fragmented, and GPU’s performance gain could be amortized by the frequent kernel switches between enclave and GPU. When θ increases, the latency gain became more stable because the partitioned operators became less fragmented, thus fewer kernel switches would occur.

Comparing figure 6a with figure 6b, we observed that SOTER achieved more stable performance on Transformer. This is because, the number of operators in VGG19 is relatively limited (only 41 associative operators with 18 linear operators), while the Transformer has a rich set of operators (360 associative operators with 311 linear operators). Hence, the randomness of selecting associative operators with different computation volumes was downgraded, leading to a more stable performance. This implies that SOTER is more suitable for big complex models with a rich set of operators.

6.3 Security analysis

As mentioned in §3.2, SOTER provides *semantic security* guarantee for protecting model confidentiality: knowing only ciphertexts of parameter-morphed operators, it is infeasible

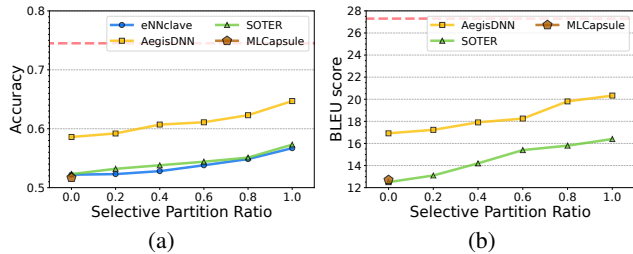


Figure 7: The model stealing experiments of all baseline systems running (a) VGG19 and (b) Transformer. The red dotted line marks the ground-truth accuracy/ BLEU score of VGG19 and Transformer respectively.

for a computationally-bounded adversary to learn about operators’ *exact* blinding coins hidden in enclave. It is notable that increasing the number of partitioned morphed operators (with a larger θ) does not degrade the confidentiality because (1) SOTER generates coins for each operator independently in enclave, and (2) an execution result of a morphed operator is concealed to ensure that it reveals neither the coin of that operator, nor the coins of all previous operators (§4.1).

Quantifying information leakage. To evaluate whether our confidentiality analysis given above has a meaningful effect in practice, we applied model stealing attacks [35, 47] on all baseline systems to quantify how much information could leak with that system. A model stealing attack feeds synthetic data to a *victim model* to get an output, and trains a *substitute model* (depicted as Sub_M) with these new samples to mimic the inference behavior of the *victim model*.

- *Setup.* We targeted two popular models, VGG19 and Transformer, running on all baseline systems. We did not run Transformer on eNNclave because eNNclave does not support such a case. We conducted the state-of-the-art Wang’s attack [67] for model stealing. Concretely, for the training dataset, with the bounded computational capability assumption [35], we generated synthetic data which composes 10% of total training samples; for the backbone of the substitute model, we adopted VGG13 as the architecture for VGG19 and the standard encoder/decoder architecture for Transformer. We initialized all unknown parameters using a truncated normal distribution with std of 0.02 and learning rate of 0.0001. By training on one Nvidia 2080TI GPU, Sub_{VGG} converged within 150 epochs and Sub_{TF} converged within 13 epochs.

- *Results.* Figure 7 depicts the inference results of the trained substitute model on all baselines. MLCapsule, which shields the entire model within enclave (i.e., $\theta = 0$), achieved the minimum accuracy through the stealing attack. The accuracy results of AegisDNN grew as θ increased, because AegisDNN only runs partial model layers in enclave and exposes other layers with plaintexts to the adversary. With more plaintext revealed in AegisDNN, the accuracy of Sub_{VGG} and BLEU of Sub_{TF} increased dramatically. SOTER, however, achieved comparable results as eNNclave, which directly replaces partitioned operators’ parameters with insensitive public parame-

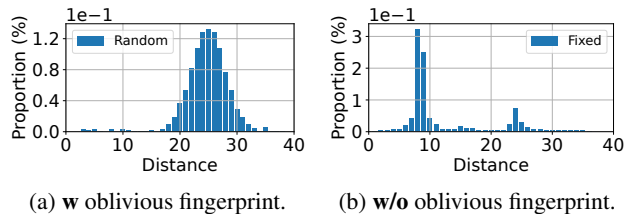


Figure 8: The l_2 distance distribution of randomly sampled fingerprints with or without oblivious fingerprinting technique.

ters. Notably, this implied that SOTER’s MTR protocol with semantic security guarantee is effective to provide sufficient model confidentiality.

6.4 Robustness to integrity breaches

In this subsection, we begin by investigating the obliviousness of SOTER-generated fingerprints for obfuscating the adversary. Then we evaluated SOTER’s robustness to integrity breaches of partitioned operators outside enclave.

Figure 8 shows that, when running with SOTER’s oblivious fingerprinting technique, the l_2 distance between randomly sampled fingerprints shares the same form of normal distribution as normal query input (Figure 8a). Hence the fingerprints are oblivious to the adversary. Whereas, when running with fixed fingerprints (Figure 8b), the distribution dramatically changed, hence those fixed fingerprints are observable to the adversary. We evaluated the robustness of SOTER under 10K random perturbation attacks, which randomly perturbs the parameters of partitioned operators. All attacks were successfully detected, and 99.9% attacks were detected with less than ten fingerprints while we incurred zero false positives.

7 Discussion

Model leakage from black-box attacks. Modern black-box attacks [4, 35, 47, 50] use inference APIs of an inference service to learn private information of an offloaded model or even the training dataset, by using the inference results of perturbed input queries. These attacks widely exist in either cloud-side or client-side inference systems [27, 36, 44, 56], as inference systems inevitably open access to arbitrary queries.

Currently, there is no general defense against black-box attacks other than query authentication and rate limiting [35, 47]. But fortunately, defenses against specific attacks are emerging. For instance, Prada [35] uses query distance auditing to mitigate model extraction attacks with adversarial examples; Shokroi et al. [60] use differential privacy to train DNNs that do not leak model owner’s sensitive training datasets.

The guarantees of SOTER, like all prior secure inference systems targeting at protecting the model at the edge or the cloud, are complementary to those provided by any defenses. With sufficient engineering effort, these mitigations can be integrated into SOTER to provide even stronger privacy guarantees, and we leave it to future work.

Computation overflow. Multiplying the model parameter with SOTER’s blinding coin (§4.1) may, in rare cases, causes

potential computation overflows (we did not find any computation overflow in our experiments in §6). However, even if an overflow occasionally happens, SOTER can immediately detect such overflow by checking if the computed result is INF or -INF [39]. Then, SOTER can simply re-morph the overflowed GPU operator by replacing the blinding coin with a smaller new coin, which can be prepared offline (§4.1).

Trusted GPU. Although there are some promising trusted GPU research systems [3, 32, 46, 65] that can protect model confidentiality and accelerate inference with the strong GPU computing power, these systems are not publicly available as they either require extensive hardware modifications [65] or support only hardware simulators [3, 32, 46]. Thus, in this paper, we consider only publicly available (untrusted) GPU deployments.

Limitation. SOTER requires clients at the edge to be equipped with TEE (e.g., Intel SGX) due to the lack of commonly available trusted GPUs [3, 65]. SGX has been pervasively used in existing secure inference systems [27, 56] and is commonly available in modern Intel CPUs. While the inherited SGX vulnerabilities of sophisticated side-channel attacks based on timing or cache access patterns still exist [18, 59, 66], we do not consider these attacks currently, and we believe these attacks can be mitigated by integrating with state-of-the-art defenses [18, 19, 55].

8 Conclusion

We present SOTER, the first secure inference system that ensures model confidentiality, low latency, high accuracy with integrity protection for general DNN models. SOTER's MTR protocol carries out cooperative executions between the TEE and GPU. Specifically, SOTER morphs a fraction of associative operators' parameters to run on a GPU, SOTER conceals the execution results on GPU and then restores the real execution results in TEE. SOTER efficiently generates fingerprints to check the integrity of partitioned operators. Evaluation on notable client-side secure inference systems and all prevalent types of DNN models shows that, compared to existing relevant baselines, SOTER achieved comparable strong confidentiality as the TEE-shielding approach, comparable low-latency as the partition-based approach, high accuracy same as insecure inference, and overwhelming high probability of detecting integrity breaches of any partitioned operators. These features make SOTER unique in encouraging enormous model providers to develop powerful models and deploy them on third-party edge devices.

Acknowledgments

We thank the anonymous shepherd and all anonymous reviewers for their helpful comments. This work is supported in part by a Huawei Flagship Research Grant in 2021, a HKU-SCF FinTech Academy R&D Funding Scheme in 2021, HK RGC GRF (17202318, 17207117), HK ITF (GHP/169/20SZ), the Pujiang Lab (Heming Cui is a courtesy researcher in this lab),

the HKU and IS-CAS Joint Lab for Intelligent System Software, Hong Kong RGC Project No. PolyU15223918, the Science, Technology and Innovation Commission of Shenzhen Municipality under Grant No.SGDY20201103095408029, and National Natural Science Foundation of China under Grant No.62002151.

References

- [1] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.
- [2] Anshul Aggarwal, Trevor E Carlson, Reza Shokri, and Shruti Tople. Soteria: In search of efficient neural networks for private inference. *arXiv preprint arXiv:2007.12934*, 2020.
- [3] Aref Asvadishrehjini, Murat Kantarcioglu, and Bradley Malin. Goat: Gpu outsourcing of deep learning training with asynchronous probabilistic integrity verification inside trusted execution environment. *arXiv preprint arXiv:2010.08855*, 2020.
- [4] Buse Gul Atli, Sebastian Szyller, Mika Juuti, Samuel Marchal, and N Asokan. Extraction of complex dnn models: Real threat or boogeyman? In *International Workshop on Engineering Dependable and Secure Machine Learning Systems*, pages 42–57. Springer, 2020.
- [5] J Aumasson and L Merino. Sgx secure enclaves in practice—security and crypto review. *Black Hat*, 2016.
- [6] Gregory V Bard. The vulnerability of ssl to chosen plaintext attack. *IACR Cryptol. ePrint Arch.*, 2004:111, 2004.
- [7] Mihir Bellare, Stefano Tessaro, and Alexander Vardy. Semantic security for the wiretap channel. In *Annual Cryptology Conference*, pages 294–311. Springer, 2012.
- [8] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Praseoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.
- [9] Adith Boloor, Karthik Garimella, Xin He, Christopher Gill, Yevgeniy Vorobeychik, and Xuan Zhang. Attacking vision-based perception in end-to-end autonomous driving models. *Journal of Systems Architecture*, 110:101766, 2020.
- [10] Han Cai, Ligeng Zhu, and Song Han. Proxlessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

- [11] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 210–227, 2021.
- [12] Xusheng Chen, Shixiong Zhao, Ji Qi, Jianyu Jiang, Haoze Song, Cheng Wang, Tsz On Li, TH Hubert Chan, Fengwei Zhang, Xiapu Luo, et al. Efficient and dos-resistant consensus for permissioned blockchains. *Performance Evaluation*, 153:102244, 2022.
- [13] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.
- [14] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.
- [15] Piali Das, Nikita Ivkin, Tanya Bansal, Laurence Rousnel, Philip Gautier, Zohar Karnin, Leo Dirac, Lakshmi Ramakrishnan, Andre Perunicic, Iaroslav Shcherbatyi, et al. Amazon sagemaker autopilot: a white box automl solution at scale. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, pages 1–7, 2020.
- [16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.
- [17] Shuiguang Deng, Hailiang Zhao, Weijia Fang, Jianwei Yin, Schahram Dustdar, and Albert Y Zomaya. Edge intelligence: The confluence of edge computing and artificial intelligence. *IEEE Internet of Things Journal*, 7(8):7457–7469, 2020.
- [18] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. Hybcache: Hybrid side-channel-resilient caches for trusted execution environments. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 451–468, 2020.
- [19] Anuj Dubey, Rosario Cammarota, Vikram Suresh, and Aydin Aysu. Guarding machine learning hardware against physical side-channel attacks. *arXiv preprint arXiv:2109.00187*, 2021.
- [20] S Durga, Esther Daniel, and S Deepakanmani. Dnn-based decision support system for ecg abnormalities. In *2nd EAI International Conference on Big Data Innovation for Sustainable Cognitive Computing*, pages 331–338. Springer, 2021.
- [21] Tarek Elgamal and Klara Nahrstedt. Serdab: An iot framework for partitioning neural networks computation across multiple enclaves. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 519–528. IEEE, 2020.
- [22] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.
- [23] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *SC’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2012.
- [24] Alessandro Giusti, Dan C Cireşan, Jonathan Masci, Luca M Gambardella, and Jürgen Schmidhuber. Fast image scanning with deep max-pooling convolutional neural networks. In *2013 IEEE International Conference on Image Processing*, pages 4034–4038. IEEE, 2013.
- [25] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.
- [26] Karan Grover, Shruti Tople, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure dnn inference with enclaves. *arXiv preprint arXiv:1810.00602*, 2018.
- [27] Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Maximilian Augustin, Michael Backes, and Mario Fritz. Mlcapsule: Guarded offline deployment of machine learning as a service. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3300–3309, 2021.
- [28] Hanieh Hashemi, Yongqin Wang, and Murali Annavaram. Darknight: An accelerated framework for privacy and integrity preserving deep learning using trusted hardware. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 212–224, 2021.
- [29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [30] David A Hensher and William H Greene. The mixed logit model: the state of practice. *Transportation*, 30(2):133–176, 2003.

- [31] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.
- [32] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous isolated execution for commodity gpus. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 455–468, 2019.
- [33] Jianyu Jiang, Xusheng Chen, TszOn Li, Cheng Wang, Tianxiang Shen, Shixiong Zhao, Heming Cui, Cho-Li Wang, and Fengwei Zhang. Uranus: Simple, efficient sgx programming and its applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 826–840, 2020.
- [34] Neil F Johnson and Sushil Jajodia. Exploring steganography: Seeing the unseen. *Computer*, 31(2):26–34, 1998.
- [35] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N Asokan. Prada: protecting against dnn model stealing attacks. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 512–527. IEEE, 2019.
- [36] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, 2018.
- [37] Eike Kiltz, Adam O’Neill, and Adam Smith. Instantiability of rsa-oaep under chosen-plaintext attack. In *Annual Cryptology Conference*, pages 295–313. Springer, 2010.
- [38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.
- [39] Ignacio Laguna. Fpchecker: Detecting floating-point exceptions in gpu applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1126–1129. IEEE, 2019.
- [40] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [41] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. Occlumency: Privacy-preserving remote deep-learning inference using sgx. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–17, 2019.
- [42] Wei-Fen Lin, Der-Yu Tsai, Luba Tang, Cheng-Tao Hsieh, Cheng-Yi Chou, Ping-Hao Chang, and Luis Hsu. Onnc: A compilation framework connecting onnx to proprietary deep learning accelerators. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 214–218. IEEE, 2019.
- [43] Xiaolong Ma, Fu-Ming Guo, Wei Niu, Xue Lin, Jian Tang, Kaisheng Ma, Bin Ren, and Yanzhi Wang. Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5117–5124, 2020.
- [44] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2505–2522, 2020.
- [45] Robert C Moore and Will Lewis. Intelligent selection of language model training data. 2010.
- [46] Lucien KL Ng, Sherman SM Chow, Anna PY Woo, Donald PH Wong, and Yongjun Zhao. Goten: Gpu-outsourcing trusted execution of neural network training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 14876–14883, 2021.
- [47] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Prediction poisoning: Towards defenses against dnn model stealing attacks. *arXiv preprint arXiv:1906.10908*, 2019.
- [48] Jeffrey Overbey, William Traves, and Jerzy Wojoydylo. On the keyspace of the hill cipher. *Cryptologia*, 29(1):59–72, 2005.
- [49] Daniele Palossi, Antonio Loquercio, Francesco Conti, Eric Flamand, Davide Scaramuzza, and Luca Benini. A 64-mw dnn-based visual navigation engine for autonomous nano-drones. *IEEE Internet of Things Journal*, 6(5):8357–8371, 2019.
- [50] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.
- [51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

- [52] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Annual cryptology conference*, pages 502–519. Springer, 2010.
- [53] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 51(6):1–36, 2019.
- [54] Ji Qi, Xusheng Chen, Yunpeng Jiang, Jianyu Jiang, Tianxiang Shen, Shixiong Zhao, Sen Wang, Gong Zhang, Li Chen, Man Ho Au, et al. Bidl: A high-throughput, low-latency permissioned blockchain framework for data-center networks. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 18–34, 2021.
- [55] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. Zerotracer: Oblivious memory primitives from intel sgx. In *NDSS*, 2018.
- [56] Alexander Schlögl and Rainer Böhme. ennclave: Offline inference with model confidentiality. In *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, pages 93–104, 2020.
- [57] Jean Senellart, Dakun Zhang, Bo Wang, Guillaume Klein, Jean-Pierre Ramatchandirin, Josep M Crego, and Alexander M Rush. Opennmt system description for wmt 2018: 800 words/sec on a single-core cpu. In *Proceedings of the 2nd Workshop on Neural Machine Translation and Generation*, pages 122–128, 2018.
- [58] AMD SEV-SNP. Strengthening vm isolation with integrity protection and more. *White Paper, January*, 2020.
- [59] Tianxiang Shen, Jianyu Jiang, Yunpeng Jiang, Xusheng Chen, Ji Qi, Shixiong Zhao, Fengwei Zhang, Xiapu Luo, and Heming Cui. Daenet: Making strong anonymity scale in a fully decentralized network. *IEEE Transactions on Dependable and Secure Computing*, 2021.
- [60] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1310–1321, 2015.
- [61] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [62] Florian Tramèr and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2018.
- [63] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017.
- [64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.
- [65] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on gpus. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 681–696, 2018.
- [66] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434, 2017.
- [67] Wenxuan Wang, Bangjie Yin, Taiping Yao, Li Zhang, Yanwei Fu, Shouhong Ding, Jilin Li, Feiyue Huang, and Xiangyang Xue. Delving into data: Effectively substitute training for black-box attack. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4761–4770, June 2021.
- [68] Yecheng Xiang, Yidi Wang, Hyunjong Choi, Mohsen Karimi, and Hyoseung Kim. Aegisdnn: Dependable and timely execution of dnn tasks with sgx. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 68–81. IEEE, 2021.
- [69] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.
- [70] Chao Zhang and Philip C Woodland. Parameterised sigmoid and relu hidden activation functions for dnn acoustic modelling. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.
- [71] Jiliang Zhang, Wuqiao Chen, and Yuqi Niu. Deepcheck: A non-intrusive control-flow integrity checking based on deep learning. *arXiv preprint arXiv:1905.01858*, 2019.
- [72] Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Sen Wang, Peng Wang, Gong Zhang, et al. v pipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel dnn training. *IEEE Transactions on Parallel and Distributed Systems*, 33(3):489–506, 2021.

IPLFS: Log-Structured File System without Garbage Collection

Juwon Kim Minsu Jang Muhammad Danish Tehseen Joontaek Oh Youjip Won
Department of Electrical Engineering, KAIST

Abstract

In this work, we develop the log-structured filesystem that is free from garbage collection. There are two key technical ingredients: *IPLFS*, a log-structured filesystem for infinite partition, and *Interval Mapping*, a space-efficient LBA-to-PBA mapping for infinite filesystem partition. In IPLFS, we separate the filesystem partition size from the physical storage size and set the size of the logical partition large enough so that there is no lack of free segments in the logical partition during SSD's lifespan. This allows the filesystem to write the updates in append-only fashion without reclaiming the invalid filesystem blocks. We revise the metadata structure of the baseline filesystem, F2FS, so that it can efficiently handle the storage partition with 2^{64} sectors. We develop Interval Mapping to minimize the memory requirement for the LBA-to-PBA translation in FTL. Interval Mapping is a three level mapping tree. It maintains mapping only for actively used filesystem region. With Interval Mapping, the FTL can maintain the mapping for the 2^{64} sector range with almost identical memory requirement with the page mapping whose LBA range is limited by the size of the storage capacity. We implement the IPLFS on Linux kernel 5.11.0 and prototype the Interval Mapping in OpenSSD. By eliminating the filesystem level garbage collection, IPLFS outperforms F2FS by up to $12.8\times$ (FIO) and $3.73\times$ (MySQL YCSB A), respectively.

1 Introduction

Log-structured filesystem [49] has become a popular storage management system due to its unique append-only nature. This append-only nature brings significant performance advantage against the journaling-based counterparts in practically all types of storage devices including hard disk [49, 50], flash storage [35], shingled magnetic recording (SMR) drive [44] and even persistent memory [55].

The append-only nature of the log-structured filesystem brings another dimension of complexity to the filesystem: the *garbage collection*. As the log-structured filesystem ages, the filesystem runs out of free segments and needs to reclaim the obsolete filesystem blocks to make

more free segments. The activity of reclaiming the invalid blocks is called *garbage collection*. When the filesystem performs the garbage collection in the foreground, it freezes the entire filesystem till it completes [2]. The garbage collection exposes the underlying log-structured filesystem under excessive tail latency and lowers the throughput of the filesystem. The garbage collection also generates extra write traffics that increase flash wears.

A fair amount of works have been dedicated to mitigate the overhead of garbage collection in the log-structured filesystem. They include performing the garbage collection in the idle period [10, 43, 47], performing the garbage collection in a preemptive way [37], selecting the right victim segment to maximize the garbage collection efficiency, e.g. greedy, cost-benefit, age, etc [27, 41]. Some works proposed to cluster the file blocks with a similar lifespan together to improve the efficiency of the garbage collection [31, 35, 48]. The overhead of device-level garbage collection is also of serious concern. A large body of works are dedicated to mitigate the overhead of the device-level garbage collection [12, 36, 61]. When the log-structured filesystem is used with flash storage, the overhead of garbage collection may compound due to the garbage collection activities at the filesystem as well as at the device [58]. A number of works proposed to enable the host filesystem to directly manage the flash pages in the storage and eliminate the device-level garbage collection [39, 59, 60]. Recently proposed ZNS (zoned name space) treats the storage device as append-only log so that address mapping and device-level garbage collection become much simpler [5, 20]. Despite all these efforts, the root cause of garbage collection still remains neglected; the filesystem needs to reclaim the invalid filesystem blocks to make the free segments.

For several decades, the operating systems have been separating the logical entity from the associated physical entity for various types of physical resources. The typical examples include virtualizing the CPU [13], virtualizing the memory [15] or virtualizing the entire computer system [11, 17]. Unlike the other physical resources, modern operating system still tightly couples the logical storage partition from the physical storage and the size of the logical partition is bounded by the capacity of the

physical storage.

In this work, we design the log-structured filesystem without garbage collection. We separate the logical partition from the physical storage and allow the filesystem to define very large logical partition independent of the capacity of the physical storage. We make the size of the logical partition large enough so that the filesystem never runs out of free segments during the lifespan of the flash storage. With very large logical partition, the filesystem does not have to reclaim the invalid filesystem blocks and therefore can eliminate the critical drawbacks of the log-structured filesystem, the garbage collection. In the absence of the garbage collection, we can greatly simplify the log-structured filesystem design.

Our work consists of two key ingredients. First is Log-structured filesystem for Infinite Partition, *IPLFS*. The second is the space-efficient LBA-to-PBA mapping, *Interval Mapping*. For IPLFS, we use F2FS as the baseline filesystem. Modern IO subsystem uses 64 bit **unsigned long** to represent the sector number. IPLFS allows that the logical storage partition for the log-structured filesystem can grow as large as 2^{64} sectors. When the filesystem writes the disk block in an append-only manner, the lifespan of flash storage is going to expire long before it reaches the end of the filesystem partition. IPLFS ensures that the number of valid blocks does not exceed the physical storage partition. IPLFS eliminates the allocation bitmap and the reverse mapping information from the log-structured filesystem. We develop *Discard Bitmap* and *Discard Logging* to discard the invalid filesystem blocks. To support prohibitively large LBA space, we develop a space efficient LBA-to-PBA mapping, Interval Mapping FTL. Interval Mapping maintains the LBA-to-PBA mapping information only for the actively used filesystem regions. The contribution of our work can be summarized as follows.

- We successfully eliminate the key complication, garbage collection, from the log-structured filesystem. We analyze the metadata structures of the log-structured filesystem and redesign the log-structured filesystem to handle very large filesystem partition.
- In the absence of garbage collection, we greatly simplify the log-structured filesystem design. We eliminate the block allocation bitmap and reverse mapping from the log-structured filesystem. We develop space-efficient and crash-safe data structures to represent the invalidated filesystem blocks that need to be discarded at the storage, *Discard Bitmap* and *Discard Log*.
- We develop space efficient mapping scheme, *Interval Mapping*. With tree-based structure, Interval Mapping maintains the mapping only for the actively used filesystem region and can handle the LBA-to-PBA mapping for 8 ZByte logical storage partition.

- We develop *fixed-region mapping* and *map node compaction* to reduce the size of the mapping tree. With map node compaction, the fixed-region mapping periodically reorganizes the structure of the map node in the Interval Mapping tree to exclude the invalidated filesystem blocks from the map node.

Via eliminating the garbage collection overhead from the log-structured filesystem, IPLFS increases the benchmark performance by up to $12.8\times$ (FIO) and $3.7\times$ (MySQL YCSB-A) against F2FS, respectively. With fixed-region mapping, the memory overhead of Interval Mapping is limited by the size of the physical storage, not by the logical partition size. The memory overhead of Interval Mapping is similar to that of page mapping.

2 Background

2.1 Flash Translation Layer

Flash Translation Layer is mainly responsible for three tasks: LBA-to-PBA mapping, the garbage collection, and the wear-leveling. All these three features are for hiding the physical characteristics of the flash storage media: inability to overwrite, asymmetry between the read latency and write latency and limited erase/write cycles. FTL maintains a table that holds the physical locations of the individual logical blocks in the storage device. This data structure is called *mapping table*. The size of the mapping table is proportional to the number of LBA's which it needs to map and again it is linearly proportional to the capacity of the storage device visible to the host. Page mapping maintains mapping information in page granularity [7]. While it exhibits superior random write performance [34], it suffers from excessive memory pressure for the mapping table. Block mapping maintains a mapping in the granularity of the flash block, e.g. 2 MByte. It reduces the memory pressure for the mapping table but it leaves the SSD under block thrashing [28, 34, 38]. Hybrid mapping applies block mapping for data blocks and page mapping for log blocks to reduce the mapping table size and to avoid block thrashing in the block mapping [18, 28, 29, 42, 45].

DISCARD (or TRIM) command [52] informs the SSD that a given set of blocks in the storage are no longer needed by the filesystem. It is proposed to prohibit the garbage collection module of FTL from blindly migrating the flash pages whose contents are invalid [25].

2.2 Lifespan of the Flash Storage

Flash storage can be erased and programmed only a limited number of times [23]. Table 1 illustrates the TBW (TeraBytes Written) of flash storage products released between 2019 to 2020. TBW is the amount of data that

Manufacturer	Model	Release	TBW
Adata	XPG Gammix S50	2019	3,600
Samsung	970 EVO Plus	2019	1,200
Patriot	Viper VPR100	2019	3,115
Sabrent	Rocket Q	2019	1,800
Samsung	S980 PRO	2020	600
Samsung	870 QVO	2020	2,880
T-Force	Cardea Zero Z340	2020	1,665
WD	Black SN850	2020	1,200
SK hynix	Gold P31	2020	750

Table 1: TBW(terabytes written) of the SSD products

Dataset	Write volume (GB/day)	Time to exhaust address space (M year)
YCSB SSD [56]	1,159	20
Systor [51]	57	422
Nexus 5 [21]	13	1,853

Table 2: Daily write volume and estimated SSD lifespan, traces are from SNIA open dataset [4]

can be written to the SSD over the lifespan of the drive. In Table 1, the largest TBW corresponds to 3.6 PByte.

The actual amount of data that is written to the storage device is much smaller than what the storage device can sustain. We compute the per-day write volume from the real IO traces [4](Table 2). They correspond to IO traces in the Key-value storage [56], SYSTOR [51], and smartphone [21]. Key-value storage engine generates the largest amount of write among the three traces; it writes 1.13 TByte per day to the storage.

The modern OS uses 64 bit, **unsigned long** type, variable to represent the sector number (LBA). With **unsigned long** type variable, the host can create the logical partition of 2^{64} sectors. The size of the logical partition corresponds to 8 ZByte. With 8 ZByte logical storage partition, the log-structured filesystem can keep appending the updated blocks at the end of the active filesystem region without reclaiming the invalid filesystem blocks. The lifespan of the underlying SSD will expire long before the filesystem reaches the end of the logical partition of 8 ZByte. For YCSB-A workload in Table 2, it will take 20 million years to exhaust the 8 ZByte filesystem partition.

2.3 F2FS, a log-structured file system

F2FS is the log-structured filesystem specifically developed for the flash storage. Despite the promising characteristics, the preceding log-structured filesystems [49, 50] have failed to be widely deployed in the commodity hardware with the prime cause for the failure being the overhead of reclaiming the invalid blocks. F2FS is the first log-structured filesystem that has gained the publicity successfully. It is the default filesystem for wide variety of Android devices ranging from smartphone to

automotive [1].

F2FS divides the filesystem partition into two areas: metadata area and data area. F2FS updates the contents in the metadata area in an in-place manner and writes the data area in an append-only manner. The metadata area contains the filesystem metadata such as a superblock, a checkpoint pack, a block allocation bitmap for each segment (a segment information table, SIT), and reverse mapping information (the file id and the file offset) for each segment (segment summary area, SSA). To avoid the wandering tree problem of the out-of-place update associated with the log-structured filesystem [8], F2FS clusters the file index blocks for all files in the filesystem together in the metadata area (a node address table, NAT) and updates it in an in-place manner. F2FS organizes the data area as a set of zones. A zone consists of a set of sections and a section consists of a set of segments. The segment is the unit of disk write and the section is the unit of garbage collection. In most (if not all) deployment of F2FS, a zone and a section consist of a single segment.

F2FS defines two block types (node and data) and three hotness levels (hot, warm, and cold) to represent the update frequency of a filesystem block. The node block corresponds to the inode or the index block of the file. There are total six combinations of block type and hotness level pair. F2FS maintains the six active segments in memory for each combination. F2FS places the blocks of the same type and temperature at the same active segment. When the active segment is full, it is flushed to the storage device. F2FS clusters the filesystem blocks with the same type and hotness level together at the flash storage. This is to reduce write-amplification caused by the device level garbage collection.

F2FS reclaims the invalid filesystem blocks either when the filesystem is idle (background garbage collection) or when there runs out of free segments (foreground garbage collection). In background and foreground garbage collection, F2FS uses cost-benefit policy [41] and greedy policy [27], respectively, in selecting the victim section.

For crash recovery, F2FS reads the most recent checkpoint pack from the disk and recovers the filesystem state with respect to the time of the most recent checkpoint. Then, F2FS scans the node area of the filesystem, and identifies the files that have been made durable via `fsync()` after the most recent checkpoint. For each such file, F2FS compares the node block at the time of checkpoint and the node block synchronized to the disk via `fsync()` and identifies the newly allocated blocks and the invalidated blocks in the file. For the newly allocated blocks, F2FS updates the associated filemap. For invalidated blocks, F2FS invalidates the block allocation bitmap in the segment information table in memory. After reconstructing the allocation bitmap, the recovery

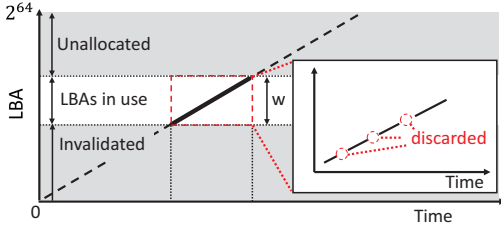


Figure 1: Active Region w in the logical partition

module creates the discard commands for the invalidated blocks.

When a filesystem operation invalidates one or more blocks, e.g. `truncate()`, or `unlink()`, F2FS updates the associated bitmap in the segment information table. F2FS checkpoints the filesystem state either periodically or when the garbage collection is triggered. In checkpoint, F2FS constructs the discard commands for the invalidated filesystem blocks. When the checkpoint finishes, F2FS wakes up the discard thread. The discard thread dispatches the discard commands in a regular interval (default = 50 msec). It limits the number of dispatch commands that are sent at a time (default = 8). It dispatches the discard command only when the system is idle.

3 Design Overview

3.1 Design Philosophy

The fundamental design philosophy behind IPLFS is the *separation of the logical storage partition from the physical storage*. The log-structured filesystem [49] offers natural ground to separate the logical partition from the physical partition due to its level of indirection inherent in the out-place update filesystem. In the legacy in-place update filesystems, a file block is bound to the fixed location in the physical storage when the file block is allocated. It is non-trivial to separate the logical partition size from the physical partition size [24]. On the other hand, the log-structured filesystem dynamically updates the file mapping information to keep track of the location of the most recent version of the file block.

The existing log-structured filesystem [32, 35] limits the size of the logical partition to the size of the associated storage device and reclaims the invalid filesystem blocks when it runs out of free blocks in the logical partition. In this work, we set the size of the logical partition large enough so that there is no lack of free LBAs during SSD’s lifespan. Fig. 1 visualizes the usage pattern of the log-structured filesystem in the very large logical partition. X and Y axes denote time and LBA, respectively. As the filesystem ages, file blocks get invalidated. The window of actively used filesystem region, w , moves towards the higher end of its filesystem partition. Actively

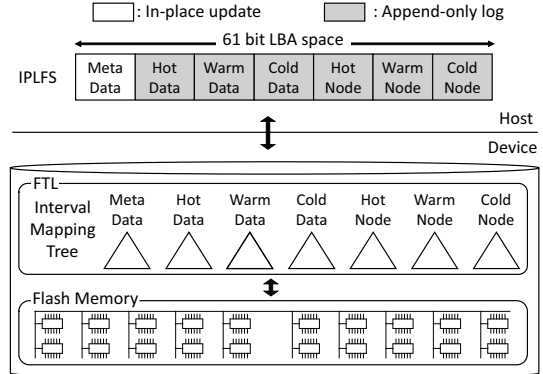


Figure 2: Concept: Log-structured filesystem for Infinite Partition, IPLFS, and Interval Mapping

used filesystem region starts at the lowest valid LBA and ends at the highest valid (allocated) LBA. Within the actively used filesystem region, w , some blocks are invalid and discarded at the storage device. When the size of the logical partition is very large, only a small fraction of the logical partition, w , is being accessed by the filesystem. The storage controller needs to maintain LBA-to-PBA mapping only for the actively used filesystem region, w .

3.2 Organization

Fig. 2 illustrates the main components of our system. Eliminating the garbage collection in the log-structured filesystem is achieved by two ingredients: Log-structured filesystem for Infinite Partition, *IPLFS* and FTL for very large logical partition, *Interval Mapping*. The first component is IPLFS. IPLFS uses F2FS as a baseline filesystem. The in-memory and on-disk structures of F2FS are carefully trimmed and modified so that it can handle the logical partition of 2^{61} blocks and that it can dispense with filesystem level garbage collection.

The second component is a space-efficient FTL, *Interval Mapping*. In most existing LBA-to-PBA mapping techniques, the number of entries in the mapping table corresponds to the number of blocks in the logical storage partition. These techniques become practically infeasible due to its prohibitive mapping table size when it needs to map 2^{61} blocks. Interval Mapping maintains an LBA-to-PBA mapping only for the actively used region in the logical storage partition. Interval Mapping is multi-level tree based mapping. By using the multi-level mapping, Interval Mapping tries to avoid allocating the mapping table entries for the invalid filesystem blocks.

4 IPLFS

IPLFS never recycles the blocks in the filesystem partition. This very nature enables IPLFS to dispense with garbage collection at the filesystem layer and yet can

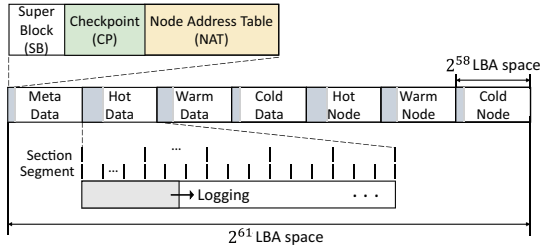


Figure 3: Multi-area Partition of IPLFS

maintain its append-only update nature. IPLFS consists of three key design ingredients: (i) multi-area partition layout, (ii) garbage collection-less metadata design, and (iii) discard map and discard logging.

4.1 Multi-area Partition Layout

We partition the entire filesystem partition of IPLFS into seven areas of the same size (Fig. 3). One area (the first one) is used for hosting the metadata of IPLFS. It holds the filesystem metadata information such as superblock and the node address table. Existing log-structured filesystems treat the filesystem partition as a single log [49, 50] or two logs [35]. IPLFS has six logs each of which accommodates the filesystem blocks of the same type and hotness level. Via clustering the filesystem blocks with similar update frequency together, IPLFS maintains the size of the actively used filesystem region small. We use MSB 3 bits of the LBA address as the area identifier. The size of each area is 2^{58} blocks, 1 ZByte.

4.2 Metadata Design

We carefully design the metadata structure of IPLFS so that it can handle the very large filesystem partition. Particular care has been taken to minimize any changes in the on-disk layout of its baseline filesystem, F2FS. Log-structured filesystem provides two essential metadata: reverse mapping and block allocation bitmap. These data structures have two main usages: the filesystem level garbage collection and the block discard. These two data structures are no longer used for the filesystem-level garbage collection because IPLFS does not perform garbage collection. IPLFS cannot use the reverse mapping and block allocation bitmap for block discard purpose, either, due to the prohibitively large logical partition. The size of the block allocation bitmap and the size of the reverse mapping information are linearly proportional to the size of the filesystem partition. Given the filesystem partition of 2^{64} sectors, 8 ZByte, the size of the block allocation bitmap and the reverse mapping corresponds to 512 PByte and 8 EByte, respectively. IPLFS cannot afford the storage space for block allocation bitmap and reverse mapping.

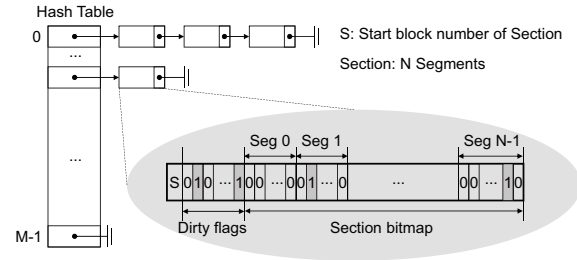


Figure 4: Discard Bitmap

IPLFS retains the node address table (NAT) in F2FS as is. The number of entries of the node address table corresponds to the maximum number of inodes in the filesystem partition. The number of inodes in the filesystem is limited by the capacity of the physical storage, not by the size of the logical filesystem partition. The size of node address table does not increase even though the size of the logical partition is very large.

In IPLFS, we remove the block allocation bitmap (Segment Information Table) and reverse mapping information (Segment Summary Area) from its baseline filesystem, F2FS and develop a new metadata structure for discarding the filesystem blocks, *Discard Bitmap* and *Discard Log*.

4.3 Discarding the Invalid Blocks

The log-structured filesystem maintains the block allocation bitmap for two reasons: to represent the space utilization of the individual segments and to keep track of the newly invalidated filesystem blocks. Former is for the filesystem-level garbage collection purpose and the latter is for discarding the filesystem blocks. In IPLFS, the former reason for maintaining the allocation bitmap disappears but the latter reason remains outstanding.

Eliminating the block allocation bitmap, we develop a new data structure, *discard bitmap*, that represents the newly invalidated filesystem blocks since the last checkpoint. IPLFS maintains the discard bitmap in per-section basis. In IPLFS, a section consists of more than one segments. A discard bitmap consists of two components: the start LBA of the section and the bitmap itself. When the filesystem invalidates the block, it sets the associated discard bit at the discard bitmap. IPLFS organizes a set of the discard bitmaps using the hash table. Fig. 4 illustrates the structure of the set of discard bitmaps. M and N correspond to the number of hash buckets in the hash table and the number of segments in a section. The hash table uses the section number as a hash key.

When a filesystem block is invalidated, IPLFS searches the hash table for the associated discard bitmap. If the discard bitmap is found, IPLFS updates the discard

bitmap with the newly invalidated block. If the associated discard bitmap does not exist, IPLFS allocates the discard bitmap for the section which the newly invalid block belongs to, and sets the associated bit of the block that needs to be invalidated. Then, IPLFS inserts the newly created discard bitmap at the hash table.

There is a trade-off between the section size and the filesystem performance. With a larger section size, the hash table for the discard bitmaps becomes larger. With a smaller section size, there exists more discard bitmaps in the hash table and the latency for searching the hash table becomes longer. Through experiment, we find that the section size of 1 GByte renders the reasonable balance between the filesystem performance and the memory pressure. In the later part of this paper, the section size is set as 1 GByte, i.e. 512 segments.

In each checkpoint, IPLFS scans the hash table and constructs the discard commands for each discard bitmap. After constructing the discard commands, it removes the discard bitmap from the hash table. IPLFS issues the discard commands periodically, e.g. in 50 msec interval. As in F2FS, IPLFS allocates a separate thread for dispatching the discard commands. IPLFS issues the discard commands in a more aggressive fashion than F2FS does. IPLFS dispatches the discard commands no matter whether there is a pending I/O or not. In F2FS, the dispatch thread issues the discard commands only when there is no pending I/O. In IPLFS, the dispatch thread issues up to sixteen discard commands each time when it wakes up. F2FS takes particular care to prohibit the discard command from interfering with the foreground IO requests [3]. We find in our platform (OpenSSD), the aggressive discard policy renders better benchmark performance since it makes the SSD garbage collection more efficient and reduces the write amplification.

4.4 Discard Logging

In the absence of the block allocation bitmap, IPLFS is subject to the *Storage Leak*. Storage Leak denotes the situation where the flash page contains invalid filesystem block and the filesystem never reclaims the associated flash page. Assume that the system crashes while there are outstanding discard commands. As a result of the system crash, the outstanding discard commands are lost. In F2FS, the recovery routine creates the discard commands based upon the recovered allocation bitmap.

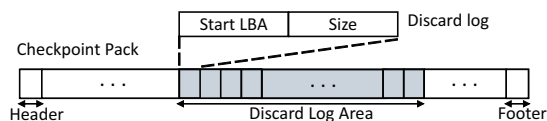


Figure 5: Checkpoint Pack with Discard Logs

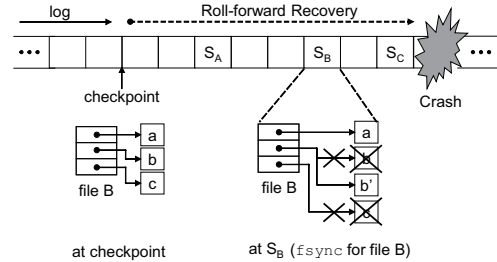


Figure 6: Roll-forward Recovery in IPLFS

Unlike F2FS, IPLFS does not have allocation bitmap and cannot reconstruct the discard commands that are lost due to crash. The flash pages associated with the lost discard commands will remain valid permanently even though they will no longer be used.

To save IPLFS from Storage Leak, we develop a mechanism called *Discard Logging*. In Discard Logging, IPLFS checkpoints the information associated with the discard commands prior to issuing the discard commands to the storage. With discard logging, IPLFS guarantees that discard command is made durable at the storage before it is issued to the storage. With Discard logging, IPLFS can recover the outstanding discard commands when the system crashes unexpectedly.

IPLFS allocates a certain region, *Discard Log Area* at the checkpoint pack (Fig. 5). At each checkpoint, IPLFS scans the discard bitmap and creates the discard commands. After it finishes creating the discard commands, IPLFS logs the information associated with the discard commands, [startLBA, Length], at the discard log area of the in-memory checkpoint pack. After it finishes preparing the checkpoint pack, it synchronizes the checkpoint pack to the disk. After the checkpoint, IPLFS wakes up the discard thread for issuing the discard commands.

When the system crashes, IPLFS recovers the discard commands in two phases. In roll-backward recovery, the recovery module reads the most recent checkpoint pack and reconstructs the discard commands with respect to the discard logs. In roll-forward recovery, IPLFS identifies the fsynced files after the most recent checkpoint. IPLFS compares the node block that is found at the roll-forward recovery phase and the node block at the time of the checkpoint. IPLFS then identifies the changes in the block allocation and updates the filemap with respect to the newly allocated node blocks. Based upon the difference on the block allocation, IPLFS identifies the invalidated filesystem blocks and constructs the discard commands for the invalidated blocks. Fig. 6 illustrates how IPLFS reconstructs the discard commands. At the time of the checkpoint, the node block of file B consists of three file blocks, a, b and c. After the checkpoint, the file block b is updated to b' and file block c is truncated in file B. Then, file B is synchronized to disk through

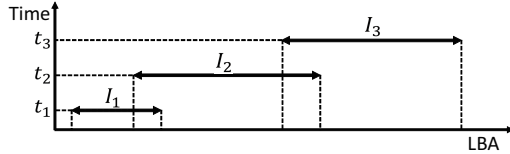


Figure 7: active region of the filesystem, I_j : active region at time t_j

`fsync()`. After `fsync()`, node block of file B refers to two blocks, a and b'. Block b' is newly allocated and block b and block c are invalidated. In roll-forward recovery, IPLFS updates the filemap of file B to refer to a and b' and creates the discard command for discarding b and c.

5 Interval Mapping

5.1 Design

We develop a space-efficient LBA-to-PBA mapping, called *Interval Mapping*. It is similar to interval tree [16] in that each leaf node has an interval of LBA's associated with it. Unlike interval tree, the height of the interval mapping tree is fixed to three. Limiting the height of the tree, the interval mapping increases the fan-out degree of the root node to accommodate the new leaf nodes. In designing the LBA-to-PBA mapping, we exploit the fact that in IPLFS, the actively used filesystem region moves towards the higher end of the logical storage partition as the filesystem ages. Fig. 7 illustrates how the active region moves with time. At t_1 , the active region corresponds to I_1 . At t_2 , the active region corresponds to I_2 .

In IPLFS, there are 2^{61} blocks in the logical storage partition. With 16 KByte flash page size, the page mapping table size for this storage partition corresponds to 4EByte. None of the existing mapping techniques such as block mapping [18], hybrid mapping [29, 34, 38, 40, 46], or superblock-based mapping [26] can reduce the mapping table size for LBA space of 2^{61} blocks to a manageable one. Interval Mapping addresses the prohibitive mapping table size requirement in IPLFS. Flash storage for IPLFS uses the page mapping for the metadata area and Interval Mapping for each of six data areas. Storage controller identifies the interval mapping tree for the incoming LBA using the most significant three bits of LBA.

Interval Mapping is organized as three level tree (Fig. 8). We limit the height of the mapping tree to three to reduce the number of memory accesses associated with the address translation. Interval Mapping organizes a storage area as an array of *zones*. The zone is an array of *mapping segments*. The size of the zone and the size of mapping segment correspond to 16 GByte and 16 MByte, respectively.

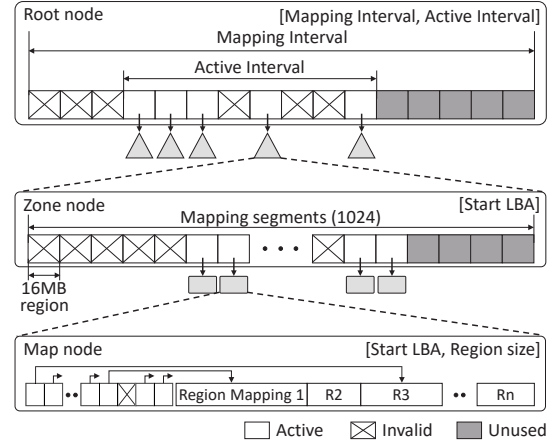


Figure 8: Structure of Interval Mapping Tree

A root node has a number of *zone nodes* as children. The sub-tree rooted at each zone node maintains a mapping for a single zone. When the interval tree is first created, the fan-out degree of the root node is set to 32. We increase the root node size to accommodate more child nodes when it is necessary. The maximum size of the root node is currently set to 1 MByte. With 1 MByte root node, the root node can have 2^{18} child nodes and can map up to 4 PByte of logical storage partition. It can be increased if the root node needs to map larger LBA region.

A single Zone node has 1024 *Map Nodes* as its child nodes. A map node maintains the LBA-to-PBA mapping for a single mapping segment. In map node, we avoid using plain table based mapping structure. Instead, for the compact mapping organization of the map node, we develop a new technique, *fixed-region mapping*, for the LBA-to-PBA mapping.

5.2 Mapping Interval and Active Interval

Interval Mapping defines two important concepts associated with mapping: the *Mapping Interval* and the *Active Interval*. Mapping Interval is a region of the logical partition that the interval mapping tree needs to map. Mapping interval is represented by the start LBA of the first zone and the start LBA of the last zone in the mapping interval, respectively. When the storage partition is created, Interval mapping creates six interval mapping trees for individual data areas of the IPLFS filesystem partition. Mapping interval is initialized when the mapping tree is first created. The start LBA of each mapping interval corresponds to the first LBA of associated filesystem area. Initially, each mapping interval consists of thirty-two zones, 512 GByte.

Active Interval is a window of actively used zones within the associated mapping interval. Active interval

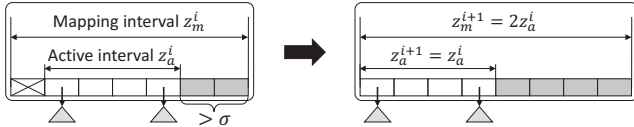


Figure 9: Updating the Mapping Interval

is similar to the active region in Fig. 7. The active interval starts at the first valid zone of the mapping interval and ends at the last valid zone of the associated mapping interval. If all filesystem blocks in a zone become invalid, the zone becomes invalid. The start of the active interval is updated to the following valid zone in the active interval when the first zone of the active interval becomes invalid. The end of the active interval is extended to the newly allocated zone if the new zone is appended to the active interval to accommodate more blocks.

As the filesystem ages, active interval moves towards the higher end of the logical partition. When the end of active interval nearly reaches the end of the mapping interval and there is little room to grow, Interval Mapping creates the new root node with the new mapping interval which can better accommodate the current active interval. The details of updating the mapping interval are as follows. First, we compute the mapping interval for the newly created root node. If the length of the current active interval is less than 1024 zones, the length of the new mapping interval is set as twice the length of the current active interval. Otherwise, it is initialized as the length of the current active interval plus sixteen zones. Second, we allocate the new root node with updated mapping interval. Third, we copy the child pointers of the old root node to the new root node.

The start of the mapping interval of the newly created root node is initialized to the start of the current active interval. The active interval of the new root node inherits the current active interval. Fig. 9 illustrates an example of creating the new root node with updated mapping interval. σ is the minimum number of free zones which the interval mapping tree needs to maintain. If the number of free zones in the mapping interval becomes less than σ , Interval Mapping updates the mapping interval creating the new root node. For an interval mapping tree, the mapping interval can be updated multiple times. When the mapping interval is updated, IPLFS allocates the new root node each time. Let i be the number of times when the mapping interval is updated for the associated mapping tree. z_m^i and z_a^i denote the number of zones in the mapping interval and the number of zones at the active interval of the i^{th} version of the root node, respectively. The new mapping interval starts at the same zone as the start of the current active interval. The length of the new mapping interval is twice the length of the current active interval, $z_m^{i+1} = 2z_a^i$. The length of the new active interval is the same as the length of the

current active interval, $z_a^{i+1} = z_a^i$.

FTL creates the new root node and updates the mapping interval in non-blocking way so that it can minimize the interference with the foreground IO request for address translation or for allocating the new zone. When a new zone node needs to be inserted at the root node while the new root node is being created, the newly created zone node is appended at the new root node and the active interval of the new root node is updated accordingly. For address translation, FTL uses the old root node if the incoming LBA belongs to the active interval of the old root node. Otherwise, it uses the new root node for address translation.

5.3 Fixed-Region Mapping

A Map Node maintains the mapping for single mapping segment, 16 MByte by default. The total size of the map nodes accounts for 99.9% of the entire mapping tree. It is critical that map node data structure is carefully designed to minimize the memory requirement as well as the mapping latency in Interval Mapping. To address the two objectives, we develop a new mapping technique called *fixed-region mapping*. Fixed-region mapping partitions a mapping segment into the same size *regions* with a given *region size*. Map node maintains the LBA-to-PBA mapping in per-region basis. Fixed-region mapping allocates the mapping table only for the valid regions, i.e. the region that has one or more valid blocks. Map node maintains a region directory which has the location of the per-region mapping tables. If the region is invalid, the associated entry in the region directory is NULL. To reduce the size of mapping table, each mapping table specifies the start LBA of the active region and excludes the mapping for invalid blocks at the beginning of the associated region.

The *region size* plays a key role in the mapping efficiency of the map node. Mapping efficiency is the ratio of the number of the valid mapping entries against the total number of mapping entries. As the region size gets smaller, the mapping segment is partitioned into smaller regions and the number of invalid regions is likely to increase. As the region size becomes smaller, the mapping efficiency improves but the region directory becomes larger. As the region size becomes larger, the mapping segment consists of smaller number of regions. With larger size region, the region directory becomes smaller but the mapping efficiency becomes worse. We need to find the right region size that can maximize the mapping efficiency and minimizes the map node size. Interval Mapping sets the region size as *the size of the smallest hole in the mapping segment*. To avoid that the region size becomes too small, we set the minimum region size, 256 KByte. When the region size corresponds to the size of the smallest hole, it is guaranteed that there can be

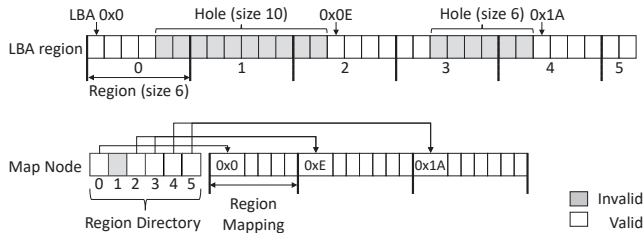


Figure 10: Mapping Segment and Map Node

only one active region in the associated region. Active region is a consecutive array of valid blocks in a region.

A map node consists of the three components: the range of the mapping segment, the region directory, and the array of mappings for individual regions. The number of entries in the region directory corresponds to the number of regions of the map node. The mapping information for each region consists of the start address of the active region and associated LBA-to-PBA mapping. Fig. 10 illustrates the mapping segment and the organization of the associated map node. There are 32 blocks in the mapping segment. There are two holes with 10 blocks and 6 blocks, respectively. The region size of this map node is set to 6. Map node partitions the mapping segment into six regions. In Region 0, there are four valid blocks. Region 1 does not have any valid blocks. The directory entry for region 1 is NULL since it does not have any valid blocks. There are three active regions in the mapping segment. The second active region spans across region 2 and region 3. The map node allocates a single region map for the active region that spans region 2 and region 3.

Interval Mapping periodically reorganizes the map node. We call it *map node compaction*. It updates the region size for the mapping segment and reconstructs the per-region mapping tables with respect to the updated regions. This is to reduce the map node size by eliminating the mappings for the invalid flash pages. When the map node is first created, the region size is set to size of the mapping segment. When the FTL invalidates the mapping table entry at the map node, it examines the mapping efficiency. If the mapping efficiency becomes smaller than a certain threshold, (50%), the FTL inserts the map node to the compaction candidate list. The compaction thread periodically scans the compaction candidate list (default 30s), and estimates the size of the reorganized map node with the updated region size. If the map node can become smaller by more than 30% after compaction, FTL reorganizes the map node. Fig. 11 illustrates the map node compaction. Before the compaction, the original map node has a single region that has 16 mapping entries and three active regions. There are two holes of four blocks and five blocks. For compaction, the region size is updated to four (the minimum

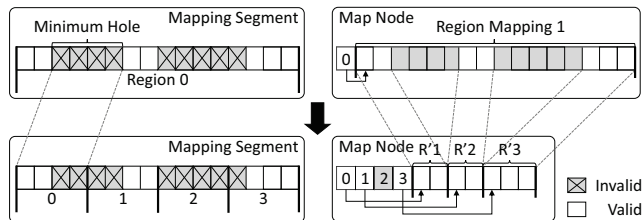


Figure 11: Reorganizing the Map Node

hole size). The mapping region is partitioned into four regions based upon the new region size, four. After compaction, the map node has three region mappings, each of which includes two, two and three mapping entries, respectively. As a result of map node compaction, the map node size decreases by approximately half.

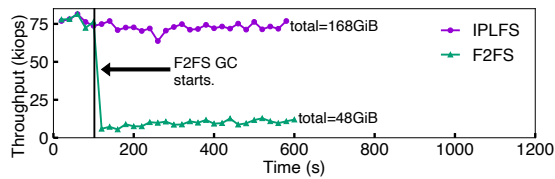
The memory overhead of Interval mapping is almost the same as the memory overhead for page mapping. Assume that the storage size is 512 GByte and flash page size is 16 KByte. The page mapping consists of a 4 KByte block bitmap (for subpage mapping [30]) and a mapping table. The page mapping requires 144 MByte (16 MByte for bitmap and 128 MByte for mapping table). The interval mapping consists of a root node, 32 zone nodes, and 2^{15} map nodes with sizes 128Byte, 4096Byte, and 4624Byte, respectively. The size of Interval Mapping corresponds to 144.6 MByte.

6 Evaluation

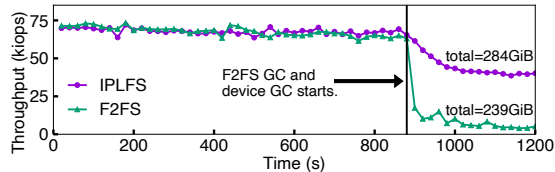
We implement IPLFS on F2FS (Linux 5.11.0) and Interval Mapping on OpenSSD (230GByte, 8 channels) [33], respectively. A default FTL in OpenSSD uses page mapping and maps LBA to PBA of different channels in a round-robin way. So does Interval Mapping FTL. We use a PC server with Intel CPU i7-4770K (3.50GHz, 4 cores), and 8 GByte DRAM for the experiment.

6.1 Eliminating the Garbage Collection

FIG 1. We examine the performance benefit of eliminating the filesystem-level garbage collection. We use FIO [6]. The logical storage partition is 30 GByte. In this experiment, four threads perform random write on 28GByte file. We measure the throughput in every 2 sec. Since the F2FS partition is almost full at the beginning of the experiment, it quickly runs out of free segment. On the other hand, the logical partition size is set to be much smaller than the physical storage size. This is to prohibit the storage device from running FTL garbage collection. The effect of eliminating the filesystem level garbage collection is substantial. In Fig. 12a, the performance of F2FS drops to 1/10 when it starts to perform garbage collection. IPLFS performance remains steady at its full speed till the experiment finishes.



(a) Only with F2FS garbage collection. Time: 600s, file size: 28GByte, partition size: 30GByte



(b) F2FS garbage collection and FTL garbage collection. Time: 1200s, file size: 210GByte, partition size: 230GByte.

Figure 12: FIO (random write) Throughput: IPLFS vs. F2FS. The number indicates the total write volume.

FIO 2. We examine the performance impact of filesystem-level garbage collection as well as device-level garbage collection. Fig. 12b illustrates the result. We set the size of the logical partition to 230 GByte, which is the physical storage capacity of OpenSSD. We perform the random write on 210 GByte file. When F2FS starts garbage collection, the throughput decreases to nearly 1/10. While IPLFS is free from filesystem level garbage collection, the underlying flash storage is not. When OpenSSD starts device-level garbage collection, the FIO performance of IPLFS decreases to 60%. The filesystem-level garbage collection bears more significant impact on the benchmark performance than the device-level garbage collection does.

MySQL. We run YCSB-A workload with MySQL and examine how database operations are interfered by the filesystem garbage collection. YCSB A workload [14] consists of the same amount of reads and updates. To

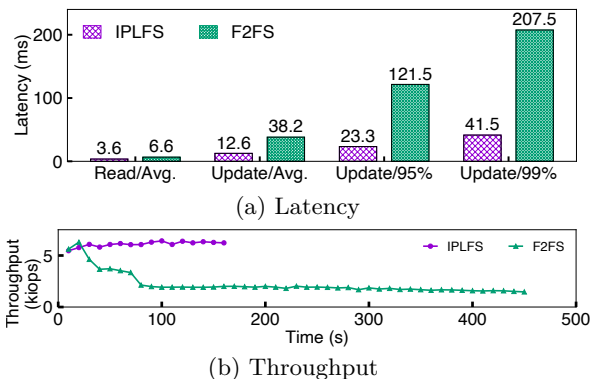


Figure 13: MySQL latency and throughput: YCSB-A, record size: 1KB, record count: 5M, operation count: 1M (read:update=1:1), threads: 50, partition size: 18GB.

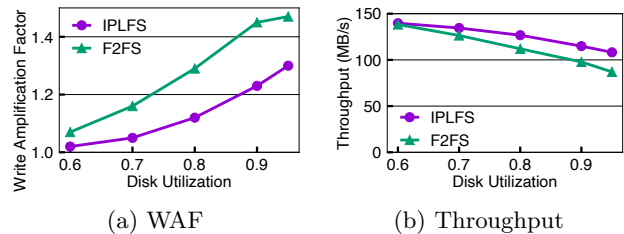


Figure 14: IPLFS vs. F2FS: filesystem benchmarks under varying disk utilization, file size: 2 MByte, partition size: 230GByte

quickly trigger the filesystem level garbage collection, the filesystem is 90% full at the beginning of the experiment. Fig. 13a illustrates the average read and update latencies of IPLFS and F2FS, respectively. The average read latency and the average update latency of IPLFS are 1/2 and 1/3 of those of F2FS, respectively. The absence of garbage collection improves the tail latency of the filesystem significantly. The tail latencies of update at 95% and at 99% in IPLFS are $5.2\times$ and $5\times$ lower than those of F2FS. Fig. 13b illustrates the throughput. IPLFS's throughput remains steady throughout the experiment. F2FS renders the similar performance to IPLFS at the beginning but the performance decreases substantially when it starts running the garbage collection.

6.2 Discard Policy of IPLFS

We examine how the more aggressive discard policy affects the FTL garbage collection and the application performance. We use filesystem workload in Filebench [53], where 50 threads create, update and delete 2 MByte files. Fig. 14a shows the write amplifications in IPLFS and F2FS. IPLFS exhibits lower write amplification than F2FS in all disk utilizations. Fig. 14b depicts the throughput under varying disk utilization. IPLFS improves the

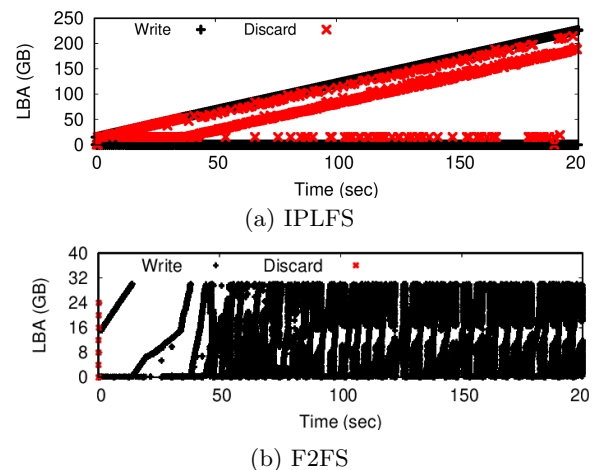


Figure 15: Block trace: Filebench filesystem workload, Partition size: 30GB

throughput by as much as 24% against F2FS. Aggressive discard policy of IPLFS saves the FTL garbage collection from migrating the invalid filesystem blocks. As a result, IPLFS renders substantial improvement in write amplification and the benchmark performance.

We examine the IO traces in IPLFS and F2FS, respectively. In IPLFS and F2FS, the logical partition sizes correspond to 1 ZByte and 30 GByte, respectively. Fig. 15a and 15b illustrate the results. IPLFS never recycles the filesystem blocks and keeps appending the blocks throughout the experiment. On the other hand, F2FS recycles the invalid filesystem blocks in round-robin manner. We observe that IPLFS issues the discard command a lot more frequently than F2FS does. This is because F2FS issues discard command only when there is no pending IO. In this experiment, F2FS rarely finds that there is no pending IO.

6.3 Address Translation Overhead

Address Translation Latency. We examine the overhead of address translation in Interval Mapping and page mapping (Fig. 16a). We run FIO with four threads. They perform random write on 10 GByte file. Interval Mapping yields 88% longer mapping latency than the page mapping. This is because Interval Mapping performs multiple index lookups for address translation. When creating a new mapping entry, Interval Mapping exhibits 3.3× longer latency than the page mapping.

End-to-end Latency. We measure the micro-benchmark performance under Interval mapping and the page mapping. Fig. 16b shows the latencies of read and write (direct IO) in FIO benchmark. The read latency and the write latency of Interval Mapping and page mapping are almost identical. This result shows that the overhead of accessing the NAND flash and the overhead of transferring the data blocks between the host and storage device account for dominant fraction of IO latency and FTL overhead is not significant.

6.4 Map Node Size

We examine the memory overhead of fixed-region mapping. We run fileserver workload with 50 threads. Each thread creates, updates and deletes 2 MByte files.

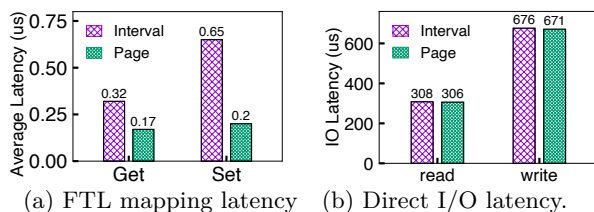


Figure 16: FTL Overhead: Interval Mapping vs. page mapping

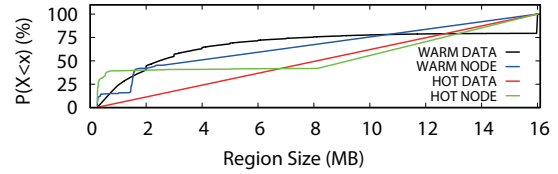


Figure 17: CDF of region size

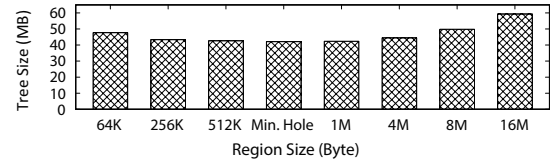


Figure 18: Mapping Tree size under varying region sizes

Region Size. We examine the IO volume associated with write and discard. We also examine the size of holes for each filesystem area (Table 3). The median hole size varies widely subject to the area type. Median hole size of the warm data area is 2860 KByte while that of the hot data area is 8 KByte. IO's for warm data area (write and discard) account for dominant fraction of all IO's (99% of write, 99% of discard).

We examine the region size distribution (Fig. 17). Region size is set as the size of the smallest hole in the associated mapping segment. For the warm data area, 3/4 of the region sizes are greater than 1 MByte. Subsequently, most of the map nodes have sixteen or less number of regions. Recall that the size of mapping segment is 16 MByte. The size of region directory accounts for approximately 1% of the map node size. In map node design, the size of the region directory is negligible.

Minimum Hole Size for region size. We compare the sizes of the mapping trees when the region size is fixed and when the region size is chosen dynamically to the size of the smallest hole in mapping segment. Fig. 18 shows results. The mapping tree becomes the smallest when we use the the minimum hole size as the region size. For small region size (64 KByte), the tree size is 13% larger than the tree size when we use the minimum hole size as region size. This is due to the increase in the region directory size. In a map node with 64 KByte region size, the region directory accounts for 13% of the total size of the mapping tables. The mapping tree size becomes larger when we use large fixed region size

Log type	W/D volume (GB)	Median (KB)	75% (KB)	90% (KB)
Warm Node	1.99 / 1.69	32	32	80
Warm Data	305.9 / 170.4	2860	6352	11764
Hot node	0.4 / 0.38	48	232	816
Hot Data	0.87 / 0.36	8	24	40

Table 3: Statistics on the hole size, Filebench fileserver workload. Average file size: 2MByte, runtime: 1600s, W/D volume: total volume of Write/Discard

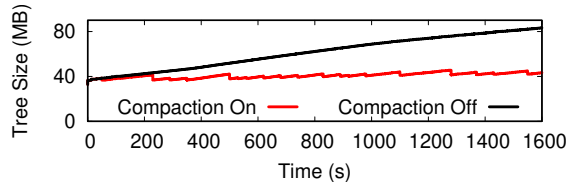


Figure 19: Mapping Tree size

(1 MB or larger) than when we use the minimum hole size as a region size. This is because with the larger region size, the fixed-region mapping fails to exclude the mappings for invalid flash pages and the mapping efficiency becomes worse.

Reorganizing the Map Node. We examine the effectiveness of map node compaction. We configure the compaction period to 30s and a compaction ratio threshold to 0.7. Total size of the files is 160GByte. When the benchmark finishes, the active interval in the filesystem partition corresponds 305GByte. Fig. 19 illustrates the sizes of mapping trees in two cases: when the Interval Mapping periodically reorganizes the map node and when it does not. Without compaction, the mapping tree size increases as the filesystem ages and reaches over 80 MByte. This is because the active interval becomes larger and the Interval Mapping allocates new map nodes as the active interval expands. With map node compaction, the mapping tree size stays at around 40 MByte. Mapping table size remains almost the same as the mapping table size for 160 GByte even though the active interval expands to 305 GByte.

7 Related Works

IO stack largely consists of two layers: the host (filesystem and block device layer) and the device (SSD). A body of works try to migrate the FTL overhead from the storage device to the host. They include DFS [24], ParaFS [60], Application Managed Flash [39], and Or-cFS [59]. In these works, the software overhead at the host side increases; the host side software directly manages the flash pages and performs essential managerial activities such as garbage collection and wear-leveling. Nameless Write eliminates the address translation layer from the IO stack [62]. Migrating the device’s functionality to the host has its cost. Flash storage needs expose its physical nature, e.g. physical flash page location, page size or block size, to the host. ZNS saves the device from the FTL overhead [9, 20]. On the contrary to these works, IPLFS aims at reducing the host side overhead, the filesystem level garbage collection with reasonable increase in the device firmware complexity.

IPLFS shares the same idea with DFS [24] in that IPLFS separates the logical filesystem partition from the physical storage capacity and fully exploits 2^{64} logical partition size. Despite of the similarity, the underlying

philosophies and the approaches of the two lie at the other ends of spectrum. DFS migrates the garbage collection from the device to the host whereas IPLFS migrates the garbage collection from the host to the device. DFS introduces new indirection layer to separate the logical partition from the physical storage. Host side’s IO stack becomes heavier to handle LBA-to-PBA mapping, garbage collection, and etc. It requires the flash device to expose physical details to the host. IPLFS does not require new layer nor any physical information of the flash storage. Separating the logical partition from the physical storage, IPLFS becomes simpler and lighter-weight than its original counter part, F2FS.

A number of works proposed to make the garbage collection more efficient. Kim et al. [31] proposed to distinguish the hot data and the cold data according to the program context. Wu et al. [54] proposed to optimize background segment cleaning scheduler based on Q-learning algorithm. Gwak et al. [19] optimized a foreground segment cleaning. A few works proposed to trigger background segment cleaning during system idle time. [10, 22, 47] Lee et al. [37] proposed preemptive garbage collection. Yan et al. [57] proposed copying valid pages in victim block to another block so that the copies handle IO operations to victim block.

To reduce the mapping table size, Kang et al. proposed to use larger granularity mapping [26]. Zhou et al. [63] increased the cache hit ratio of the page-level mapping information by employing a two-level LRU list. Liu et al. [42] proposed the FTL, which enables partial erase operation in 3D NAND flash storage.

8 Conclusion

In this work, we propose IPLFS, a log-structured filesystem for infinite partition. Separating the logical filesystem partition size from the physical storage size and making the logical filesystem partition size virtually infinite, we free the log-structured filesystem from recycling the invalid filesystem blocks. To maintain the mapping information for the prohibitively large logical filesystem partition, we develop Interval-Mapping which maintains the LBA to PBA mapping only for the actively used filesystem region. With IPLFS and Interval mapping combined together, we relieve the log-structured filesystem from the overhead of reclaiming the invalid blocks, the garbage collection.

Acknowledgements We are deeply indebted to our shepherd Youyou Lu for helping us to shape the final version of this paper. We are also grateful to the anonymous reviewers for their valuable comment and feedback. We like to thank Jay Hyun for his inspiring comment at the inception stage of this work. This work was supported by IITP, Korea (grant No. 2018-0-00549), and by NRF, Korea (grant No. NRF-2020R1A2C3008525).

References

- [1] crosshatch: switch userdata filesystem from ext4 to f2fs. <https://android.googlesource.com/device/google/crosshatch/+a0d74ba2c0b943c6370288b13ade0cf6c4868da2>.
- [2] Garbage collection semaphore in f2fs. <https://elixir.bootlin.com/linux/latest/source/fs/f2fs/f2fs.h#L1706>.
- [3] Idle checking code in the f2fs discard procedure. <https://elixir.bootlin.com/linux/v5.11/source/fs/f2fs/segment.c#L1548>.
- [4] SNIA Block I/O Traces. <http://iotta.snia.org/traces/block-io>.
- [5] Zoned Namespaces (ZNS) SSDs. <https://zonedstorage.io/introduction/zns/>.
- [6] Jens Axboe. Fio-flexible i/o tester synthetic benchmark. <https://github.com/axboe/fio>, 2005.
- [7] Amir Ban. Flash file system, U.S. Patent 5404485, Apr. 1995.
- [8] Artem B Bitvutskiy. JFFS3 design issues, 2005. <http://www.linux-mtd.infradead.org>.
- [9] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proc. of 2021 USENIX Annual Technical Conference (ATC)*, 2021.
- [10] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proc. of 1995 USENIX Technical Conference Proceedings*, 1995.
- [11] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, November 1997.
- [12] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-Time Garbage Collection for Flash-Memory Storage Systems of Real-Time Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 3(4):837–863, November 2004.
- [13] Melvin E Conway. A multiprocessor system design. In *Proc. of the fall joint computer conference (AFIPS)*, 1963.
- [14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of the 1st ACM symposium on Cloud computing*, 2010.
- [15] Robert C Daley and Jack B Dennis. Virtual memory, processes, and sharing in Multics. *Communications of the ACM*, 11(5):306–312, 1968.
- [16] H. Edelsbrunner. *Dynamic Rectangle Intersection Searching*. Forschungsberichte: Institut für Informationsverarbeitung. Inst., 1980.
- [17] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer*, 7(6):34–45, 1974.
- [18] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. *ACM Sigplan Notices*, 44(3):229–240, 2009.
- [19] Hyunho Gwak, Yunji Kang, and Dongkun Shin. Reducing garbage collection overhead of log-structured file systems with GC journaling. In *Proc. of 2015 International Symposium on Consumer Electronics (ISCE)*, 2015.
- [20] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *Proc. of 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.
- [21] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *Proc. of 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.
- [22] Martin Jambor, Tomas Hruby, Jan Taus, Kuba Krchak, and Viliam Holub. Implementation of a Linux Log-Structured File System with a Garbage Collectors. In *Proc. of ACM Special Interest Group on Operating Systems (SIGOPS)*, 2007.
- [23] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *Proc. of 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.
- [24] William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. DFS: A file system for virtualized flash storage. In *Proc. of 8th USENIX Conference on File and Storage Technologies (FAST)*, 2010.

- [25] Dong Hyun Kang and Young Ik Eom. iDiscard: enhanced Discard () scheme for flash storage devices. In *Proc. of IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2018.
- [26] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superblock-based flash translation layer for NAND flash memory. In *Proc. of the 6th ACM & IEEE International conference on Embedded software (EMSOFT)*, 2006.
- [27] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *Proc. of USENIX Technical Conference (TCO)*, 1995.
- [28] Bum Soo Kim and Gui Young Lee. Method of driving remapping in flash memory and flash memory architecture suitable therefore, U.S. Patent 6381176, Apr. 2002.
- [29] Jesung Kim, Jong Min Kim, S.H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, August 2002.
- [30] Jung-Hoon Kim, Sang-Hoon Kim, and Jin-Soo Kim. Subpage programming for extending the lifetime of NAND flash memory. In *Proc. of 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015.
- [31] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. Fully automatic stream management for multi-streamed ssds using program contexts. In *Proc. of 17th USENIX Conference on File and Storage Technologies (FAST)*, 2019.
- [32] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *ACM Special Interest Group on Operating Systems (SIGOPS)*, 40(3):102–107, July 2006.
- [33] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems. *ACM Transactions on Storage (TOS)*, 16(3):1–35, July 2020.
- [34] Hunki Kwon, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H Noh. Janus-FTL: Finding the optimal point on the spectrum between page and block mapping schemes. In *Proc. of the 10th ACM international conference on Embedded software (EMSOFT)*, 2010.
- [35] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proc. of 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.
- [36] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, and Jongman Kim. Preemptible I/O Scheduling of Garbage Collection for Solid State Drives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(2):247–260, 2013.
- [37] Junghee Lee, Youngjae Kim, Galen M Shipman, Sarp Oral, Feiyi Wang, and Jongman Kim. A semi-preemptive garbage collector for solid state drives. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.
- [38] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3), 2007.
- [39] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *Proc. of 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.
- [40] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *ACM Special Interest Group on Operating Systems (SIGOPS)*, 42(6):36–42, 2008.
- [41] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *ACM Communications*, 26(6):419–429, 1983.
- [42] Chun-yi Liu, Jagadish Kotra, Myoungsoo Jung, and Mahmut Kandemir. PEN: Design and evaluation of partial-erase for 3d nand-based high density ssds. In *Proc. of 16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.
- [43] Jeanna Neefe Matthews, Drew Roselli, Adam M Costello, Randolph Y Wang, and Thomas E Anderson. Improving the performance of log-structured file systems with adaptive methods. *ACM Special Interest Group on Operating Systems (SIGOPS)*, 31(5):238–251, 1997.
- [44] A. Palmer. SMR in Linux Systems. In *Proc. of 2020 Linux Storage and Filesystems Conference (VAULT)*, 2020.

- [45] Yubiao Pan, Yongkun Li, Huizhen Zhang, Hao Chen, and Mingwei Lin. GFTL: Group-level mapping in flash translation layer to provide efficient address translation for NAND flash-based SSDs. *IEEE Transactions on Consumer Electronics (TCE)*, 66(3):242–250, April 2020.
- [46] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(4):1–23, July 2008.
- [47] Dongil Park, Seungyong Cheon, and Youjip Won. Suspend-aware segment cleaning in log-structured file system. In *Proc. of 7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2015.
- [48] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Jooyoung Hwang, Sangyeun Cho, Daniel DG Lee, and Jaehoon Jeong. FStream: Managing flash streams in the file system. In *Proc. of 16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.
- [49] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, February 1992.
- [50] Margo I Seltzer, Keith Bostic, Marshall K McKusick, Carl Staelin, et al. An Implementation of a Log-Structured File System for UNIX. In *Proc. of USENIX Winter*, 1993.
- [51] Amir Ali Semnani, Jeffrey Pham, Burkhard Englert, and Xiaolong Wu. Virtualization technology and its impact on computer hardware architecture. In *Proc. of IEEE 8th International Conference on Information Technology: New Generations (ITNG)*, 2011.
- [52] Frank Shu. Data set management commands proposal for ata8 acs2. <https://studylib.net/doc/7497677/non-volatile-cache-command-proposal-for-ata8-acs>.
- [53] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX Login*, 41(1):6–12, 2016.
- [54] Chao Wu, Cheng Ji, and Chun Jason Xue. Reinforcement learning based background segment cleaning for log-structured file system on mobile devices. In *Proc. of 2019 IEEE International Conference on Embedded Software and Systems (ICCESS)*, 2019.
- [55] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proc. of 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.
- [56] Gala Yadgar, MOSHE Gabel, Shehbaz Jaffer, and Bianca Schroeder. SSD-based Workload Characteristics and Their Performance Implications. *ACM Transactions on Storage (TOS)*, 17(1):1–26, 2021.
- [57] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proc. of 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.
- [58] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Tagalala, and Swaminathan Sundararaman. Don't Stack Your Log On My Log. In *Proc. of USENIX 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014.
- [59] Jinsoo Yoo, Joontaek Oh, Seongjin Lee, Youjip Won, Jin-Yong Ha, Jongsung Lee, and Junseok Shim. Or-cFS: Orchestrated file system for flash storage. *ACM Transactions on Storage (TOS)*, 14(2):1–26, 2018.
- [60] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. ParaFS: A log-structured file system to exploit the internal parallelism of flash devices. In *Proc. of 2016 USENIX Annual Technical Conference (ATC)*, 2016.
- [61] Qi Zhang, Xuandong Li, Linzhang Wang, Tian Zhang, Yi Wang, and Zili Shao. Lazy-RTGC: A Real-Time Lazy Garbage Collection Mechanism with Jointly Optimizing Average and Worst Performance for NAND Flash Memory Storage Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 20(3):1–32, 2015.
- [62] Yiyang Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Deindirection for flash-based SSDs with nameless writes. In *Proc. of 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.
- [63] You Zhou, Fei Wu, Ping Huang, Xubin He, Changsheng Xie, and Jian Zhou. An efficient page-level FTL to optimize address translation in flash memory. In *Proc. of the 10th European Conference on Computer Systems (EuroSys)*, pages 1–16, 2015.

A Artifact Appendix

Abstract

Our artifact consists of two parts: IPLFS and Interval Mapping FTL. IPLFS is a log-structured filesystem with Infinite logical partition. Interval Mapping FTL is flash translation layer that maintains mapping for Infinite logical partition.

Scope

This artifact can be used to validate all experiments that measure throughput, latency and block trace in the paper.

Contents

IPLFS artifact is implemented on the top of F2FS in Ubuntu 5.11.0. The IPLFS artifact does not conduct garbage collection, and does not have metadata for garbage collection, such as a block allocation bitmap (a segment information table) and reverse mapping information (segment summary area). To replace the block allocation bitmap, discard bitmap is implemented in the IPLFS artifact. The IPLFS artifact also conducts discard logging to prevent storage leak. As specified in the paper, the IPLFS artifact partitions infinite logical space into seven areas.

Interval Mapping artifact is implemented on the top of OpenSSD. The design of the Interval Mapping artifact is three level tree, as written in the paper. The root node, zone node, and map node are all implemented in the artifact. We implement Expansion of root node and Compaction of map node in the artifact. In the artifact, there are total seven Interval Mapping trees to support for multi-area partition layout of IPLFS.

Hosting

The IPLFS artifact is uploaded in Github repository, <https://github.com/ESOS-Lab/IPLFS>. A branch named 'IPLFS-stable' contains IPLFS source code, and IPLFS format utility (f2fs-tools). A branch named 'original_kernel' is vanilla kernel which is compared with IPLFS for experiment in the paper.

The Interval Mapping artifact is uploaded in Github repository, https://github.com/ESOS-Lab/Interval_Mapping. A branch named 'main' is the artifact mainly used for the experiments in the paper. If you try measuring 'get' and 'set' latencies of Interval Mapping FTL, please use branches named 'exp-getlatency' and 'exp-setlatency', which print out the latencies of 'get' operation and 'set' operation, respectively. A branch named 'expansion+compaction'

is Interval Mapping that conducts root node expansion and map node compaction in the foreground.

Requirements

Interval Mapping FTL is built on Cosmos+ OpenSSD, the PCIe-based SSD platform on which open source SSD firmware can be developed. To operate the Interval Mapping artifact, OpenSSD is required.

Vigil-KV: Hardware-Software Co-Design to Integrate Strong Latency Determinism into Log-Structured Merge Key-Value Stores

Miryeong Kwon¹, Seungjun Lee¹, Hyunkyu Choi¹, Jooyoung Hwang², Myoungsoo Jung¹
Computer Architecture and Memory Systems Laboratory,
Korea Advanced Institute of Science and Technology (KAIST)¹, Samsung²
<http://camelab.org>

Abstract

We propose *Vigil-KV*, a hardware and software co-designed framework that eliminates long-tail latency almost perfectly by introducing strong latency determinism. To make Get latency deterministic, Vigil-KV first enables a predictable latency mode (PLM) interface on a real datacenter-scale NVMe SSD, having knowledge about the nature of the underlying flash technologies. Vigil-KV at the system-level then hides the non-deterministic time window (associated with SSD's internal tasks and/or write services) by internally scheduling the different device states of PLM across multiple physical functions. Vigil-KV further schedules compaction/flush operations and client requests being aware of PLM's restrictions thereby integrating strong latency determinism into LSM KVs. We implement Vigil-KV upon a 1.92TB NVMe SSD prototype and Linux 4.19.91, but other LSM KVs can adopt its concept. We evaluate diverse Facebook and Yahoo scenarios with Vigil-KV, and the results show that Vigil-KV can reduce the tail latency of a baseline KV system by $3.19\times$ while reducing the average latency by 34%, on average.

1 Introduction

Log-structured Merge Key-Value stores (LSM KVs) such as RocksDB [1] and LevelDB [2] are widely adopted in diverse computing domains to handle large-scale data thanks to their simplicity, scalability, and high-performance [3–10]. LSM KVs are also used in many production environments to offer large-scale storage whose capacity is beyond main memory subsystems to latency-sensitive applications. For example, Facebook uses RocksDB as the storage engine of an SQL database, which is used for social graph processing [11–14]. This type of application considers the query latency (e.g., Get) of each social action (e.g., view profile, list friends, etc.) as a first-class citizen. In particular, managing long-tail latency of reads (and latency consistency) is a matter of meeting diverse user demands and service-level agreements (SLA) [15–17].

However, we observe that the long-tail read latency of the Facebook scenario is $10.4\times$ worse than normal read oper-

ations, making the user experience inconsistent. The main contributor to this long-tail latency is device-level SSD latency, not software or operating system (OS); the execution times of all the software, including storage stack and user application, account for only 13% of the long-tail latency (99.9th percentile). We will give a detailed analysis of this in Section 3.1. The long-tail latency mainly comes from I/O interferences caused by two different levels of internal tasks: i) *LSM KV's internal tasks* and ii) *SSD's internal tasks*. LSM KVs periodically perform internal tasks such as compaction and flushing for their persistence and effectiveness [18–21]. The compaction merges data from the lower to a higher level of their LSM tree, whereas the flush writes the in-memory buffer back to the underlying storage in securing more space to buffer and making the buffered data persistent. Since the write operations of these internal tasks exhibit long latency and often stall all incoming requests, many prior studies (e.g., TRIAD [22], PebblesDB [23], and SILK [24]) reschedule additional writes of the internal tasks and serve them at future idle times. These LSM KVs would reduce the latency inconsistency imposed by the internal tasks to some extent, but we observe that they lead to serious side-effects, which deteriorate read services and increase memory footprints significantly (cf. Section 3.2).

Even with an ideal case of abolishing all the LSM KV's internal tasks, the long-tail read latency cannot be eliminated because of SSD's internal tasks such as DRAM caching/flushing [25–30], garbage collections [31–39], and read-reclaiming [40–44]. For example, even in cases where LSM KV solely reads the underlying SSD at a certain period, it exhibits long latency on the reads since the SSD internally flushes the buffered/cached data to its backend storage. Similarly, at any given time, a garbage collection or read-reclaiming can introduce a set of reads and writes, which also prevent the incoming requests from being serviced. Note that these internal tasks are scheduled by the underlying SSD firmware, which makes the read latency behaviors non-deterministic at the user-level and increases the latency significantly (cf. Section 2.2).

In this work, we propose *Vigil-KV*, a hardware and soft-

ware co-design to eliminate the long-tail latency of LSM KVs and make their read services consistently deterministic. Vigil-KV hardware offers a scheduling interface to remove SSD's internal tasks, whereas its software is designed toward eliminating the overhead imposed by LSM KV's internal tasks without delaying compaction or flushing in-memory buffer at idle times. To this end, we advocate a *predictable latency mode* (PLM) interface, which is recently added to the standard NVMe protocol [45]. We enable the brand-new interface on a high-performance NVMe SSD and enforce the read latency deterministic on a specific time window. Obviously, PLM cannot deliver the latency consistency indefinitely since SSD's internal tasks are essential to managing the reliability and persistence of the backend's storage media. For the host's finer PLM scheduling, Vigil-KV hardware also implements NVMe's NVM set features by internally partitioning the storage volume into multiple functions.

While PLM has great potential to eliminate the long-tail latency of LSM KVs by having a closer collaboration between a host and storage, there are several constraints that have not been analyzed in the literature yet. Specifically, PLM relies two essential scheduling components, *deterministic window* (DTWIN) and *non-deterministic window* (NDWIN). DTWIN is the time window to offer predictable latency, whereas NDWIN is not. This work reveals three important characteristics of PLM, which should be considered when the host communicates with the underlying SSD to achieve the latency consistency: i) *write-free on DTWIN* ii) *fair-scheduling for PLM windows*, iii) *device lockdown constraint*. First, DTWIN can be guaranteed only if there is no write request in a DTWIN period. The reason behind this DTWIN's write-free constraint is that, even though PLM supports the latency consistency by removing SSD's internal tasks at DTWIN, it cannot completely eliminate the stalls caused by online write requests coming from clients. Second, as SSD's internal tasks should be performed at some point, the longest-serving time of DTWIN is determined at design time, and NDWIN should be preserved and appropriately scheduled before jumping in DTWIN. Lastly, the host curbs I/O requests when the underlying SSD transits from NDWIN to DTWIN. This is because the transition requires a make-ready time, which must not be interrupted by any other I/O activities (i.e., device lockdown).

Based on the restrictions that we observe, the software part of Vigil-KV classifies the requests of LSM KVs at runtime and carefully assigns them to appropriate PLM time windows through our device state scheduling. This device state and request scheduling can make the latency of client-side I/O requests deterministic and have no long-tail all the time. To this end, we introduce a PLM driver atop Vigil-KV hardware, which manages all the device states across different NVM sets but makes them visible as a single storage volume. This driver makes sure that there are always $n - 1$ NVM sets having DTWIN (where n is the total number of data NVM sets) while allowing an NVM set to schedule SSD's internal task via ND-

WIN. During the device state management, it also takes into account the fair-scheduling and lockdown constraints such that a kernel-level scheduler can focus on assigning the I/O requests based on the condition of given PLM time windows. Specifically, Vigil-KV's kernel-level scheduler packs all I/O activities coming from LSM KV's internal tasks into NDWIN (scheduled by the PLM driver), which takes the overhead of all the internal tasks off the critical path in I/O services. In addition, it makes all the incoming read requests (heading to the NDWIN-scheduled set) non-blocking, inspired by a novel memory/storage array-level technique [46–49]. The kernel-level scheduler detects the read requests targeting an NVM set (configured with NDWIN) and directly serves the corresponding data without touching it at all. Since there are $n - 1$ NVM sets with DTWIN at an any given time (invisible to the host clients), the scheduler can collect data from them within the deterministic time window and reconstruct the requested data (with the help of the PLM driver) thereby making the target SSD latency consistent constantly.

Even though the Vigil-KV driver and thread can put all the internal tasks into NDWIN and isolate them from the client reads, they unfortunately fail to meet the write-free constraint. This is because of additional writes for providing atomicity and durability at the system-level (e.g., journaling). Since these writes can interfere with the reads on DTWIN, we dedicate an NVM set for the metadata management dealing with the write-ahead log (WAL) and file system journaling. To this end, we have a minor modification of RocksDB (but other LSM KVs can adopt its concept) to give a different priority to each process based on their nature of I/O activities. This technique can address the write-free constraint and make the target SSD be in all DTWIN for client requests consistently.

We prototype Vigil-KV hardware on a 1.92TB Datacenter-scale NVMe SSD, while implementing Vigil-KV software using Linux 4.19.91 and RocksDB 6.23.0. To the best of our knowledge, this is the first paper that implements the PLM interface in a real SSD and makes the read latency of LSM KVs deterministic in a hardware-software co-design manner. We evaluate six Facebook and Yahoo scenarios, and the results show that Vigil-KV can reduce the tail latency of a baseline KV system by $3.19\times$ while reducing the average latency on Get services by 34%, on average.

2 Preliminaries

We will explain RocksDB as representative of LSM KVs in this section. We will also explain the internal tasks of LSM KV and SSD in detail and analyze the challenges imposed by those two different levels of internal tasks.

2.1 Log-Structured Merge KV Stores

Figure 1a explains the major data structure and corresponding operations of RocksDB. RocksDB maintains all information

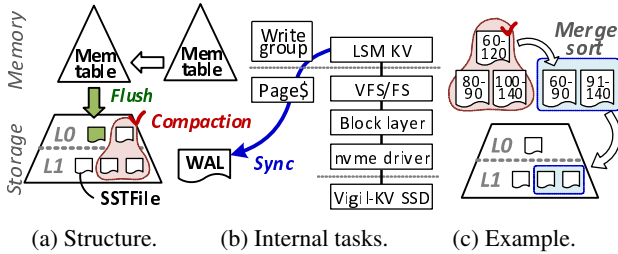


Figure 1: Log-Structured Merge KV Stores.

in the log-structured merge (LSM) tree consisting of two separate structures, each of which is optimized to volatile memory and block storage. The in-memory data structure, called *Memtable*, holds data before turning their state into persistent in an unsorted manner. Memtables allow users/clients to quickly update by serving the requests from the memory (rather than storage). The storage data structure manages key and value (KV) pairs, which are managed in an immutable form of *sorted string table* files (SSTFiles). SSTFiles are maintained in hierarchical levels, each being denoted by L0 (level-0), L1 (level-1), ..., LK (level-K).

Client operations. RocksDB supports various query services such as Put (writes), Get (reads), Delete, and Scan. Since the majority of the queries are *Put* and *Get*, this work focuses on those two operations. Users' Put requests are inserted to a Memtable as a KV pair by RocksDB if the Memtable is mutable, meaning that it has available room to update. In default, RocksDB maintains two Memtables, each taking 64MB spaces, which are the same as the size of logfiles; we will explain this in detail with LSM KV's internal tasks (i.e., flush and compaction) shortly. If there is no available space in a Memtable, RocksDB locks down and changes its state from mutable to immutable, which does not allow further updates. RocksDB then places another Memtable for the next Put requests while writing the data of the immutable Memtable to L0 by converting the Memtable to an SSTFile in the background. Since it is important to secure Memtable(s) in memory as soon as possible, turning a Memtable into L0 is performed in an unsorted and out-of-order manner. Thus, L0 can contain multiple SSTFiles associated with the same key. Later, the SSTFiles at L0 are migrated into a lower level of storage space (L1) by LSM KV's internal tasks.

On the other hand, Get requests accompany a series of reads the value associated with a given key. RocksDB first searches the key in Memtables and serves the value if there is. In cases where it fails to find the key in the Memtables, RocksDB scans all the SSTFiles residing in L0 and searches for the key. This is because the files are stored in an unsorted way, and it can be possible for L0 to have multiple SSTFiles corresponding to the given key. If RocksDB cannot find the key at L0, it goes L1 and searches again. Since L1's SSTFiles are compacted from L0, each file contains a unique key, making RocksDB faster to search the target KV pair. Note that the unsorted data structure of L0 allows RocksDB to quickly secure in-memory

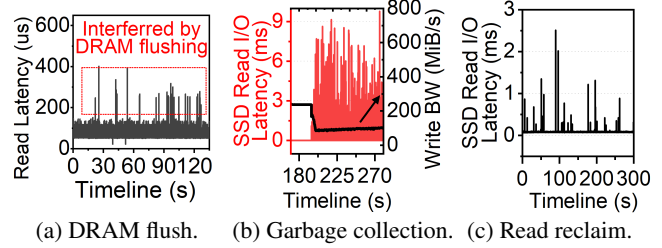


Figure 2: SSD internal tasks.

buffers, thereby preventing Put against stalls, but it introduces many storage accesses (reads) on Get services.

Internal tasks. Figure 1b illustrates the detailed procedure of LSM KV's internal tasks and major software components associated with the tasks. While Memtables are well designed toward taking performance advantage of volatile memory media, their data can be lost when there is a power failure. To make the KV pairs in the buffer persistent and durable, RocksDB writes the KV pair as a form of logfiles to a designated area in the underlying storage, called *write-ahead log* (WAL) before its Memtable update. Writing WAL (per request) is performed as a synchronous operation bypassing the page cache of the underlying file system for crash consistency control. Since it is a time-consuming task, RocksDB employs another internal buffer, called *write group* existing in front of Memtables. In the meantime, it checks the space utilization of Memtables and L0, and if there is no available space, RocksDB enqueues *flush* and/or *compaction* tasks item to reclaim a Memtable and an L0 SSTFile, respectively. These items include an appropriate pointer for the space reclaiming, which is all performed by RocksDB's background threads. For a Memtable flush, the internal task checks all the keys in a Memtable, builds an SSTFile, and flushes the SSTFile to L0. In cases of an L0 compaction, its internal task selects a target SSTFile. Consider Figure 1c as an example, the SSTFile's key ranges from 60 to 120. The task also picks L1's SSTFiles whose keys are associated with the compaction target's keys (e.g., two L1 SSTFiles, each having 80~90 and 100~140, in the figure). It then performs a merge sort by checking up all entries of three SSTFiles and letting only the latest information remain, which generates a new L1's SSTFile. Lastly, RocksDB synchronously writes the new SSTFile and removes the three old SSTFiles from the underlying SSD.

2.2 SSD Internal Tasks and Challenges

Internal DRAM flush. Since flash writes are slower than its reads by order of magnitude, high-performance SSDs employ a large size of internal DRAM, and their firmware buffers the writes [25–27, 50]. For example, our baseline NVMe hardware has 3GB DRAM buffering/caching data. These buffered writes are periodically flushed to the storage backend with a specific access pattern in favor of increasing bandwidth. Thus, even though there is no write at all for a certain period,

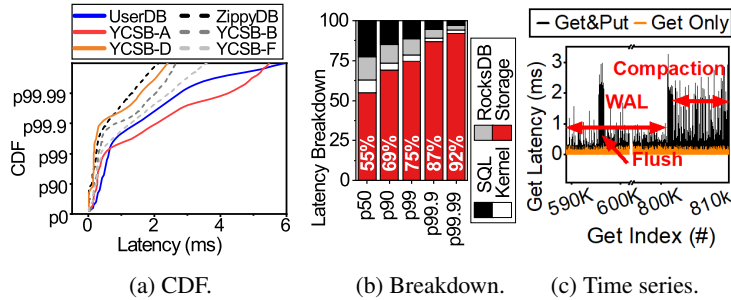


Figure 3: Long tail analysis.

draining the data (buffered previously) can interfere with incoming read operations. Figure 2a shows the read latency interfered by SSD’s internal flush; we write a block (64MB) to SSD before the test and only issue 4KB-sized read requests (in sequential) without any writes for the test period. As shown in the figure, the baseline NVMe hardware suffers from massive latency spikes, which are higher than the typical latency by $7.75\times$ at most, and its latency significantly fluctuates during the read-only time. This is because the writes introduced by the internal flush stall the reads until their service completes.

Note that, since the internal flushes are solely managed by the firmware, host software components cannot unfortunately control the latency consistency of reads. To remove the latency fluctuation analyzed above, it requires a tight collaboration between the host and firmware.

Garbage collection. Flash also has unique device-level characteristics such as erase-before-write and asymmetric I/O granularity for read/write and erase [51–54]. Because of this nature, SSD’s firmware writes incoming data into a free block (erased in advance) instead of its actual location. While this address remapping (translation) for out-of-updates makes flash compatible with the existing block devices, it needs to perform a *garbage collection* (GC) if there is no free block [31, 38, 55, 56]. Since GCs are performed on the basis of a flash block containing hundreds of pages, the valid data residing in the target block(s) should be safely migrated into a new location of a block. This internal task introduces block erase operations even longer than the flash writes and many read/writes for the migration. It exhibits long latency and stalls many incoming requests before completing the task. Figure 2b shows the read latency while performing GCs (from 195 sec). In this test, reads exhibit sustainable latency (16.2 μ s), but their latency sharply increases and reaches as high as 9.8 ms once GCs begin.

While these internal tasks significantly hamper the read performance, all their activities are essential to secure more available rooms for further requests and manage the reliability of the storage backend, which cannot be simply removed or scheduled by host-side software modules.

Read-reclaiming. Flash is very well optimized for read services at the low-level [57–60], but a read-only scenario can also introduce additional data migration and block erase operations in certain circumstances. Specifically, when one keeps

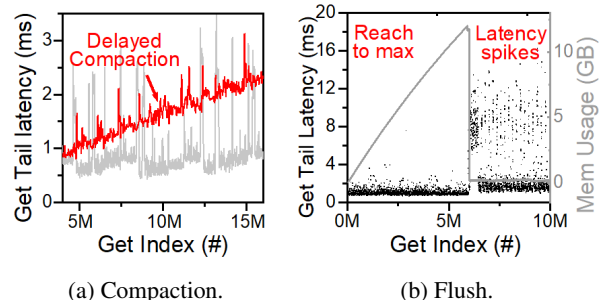


Figure 4: Limitations internal tasks.

reading out a set of pages in a block without an erase, it stresses the block even without any writes and affects all data residing in the block together. This *read disturbance* unfortunately increases error rates often beyond the coverage that parity-check codes (e.g., ECC [61–64] and LDPC [65–68]) can correct [69–72]. To address the read disturbance issue, the underlying firmware needs to periodically reset (erase) the block(s) being intensively touched over the past period. Once the firmware erases the block, its internal state returns back to the nominal state, such that the block can endure the stress imposed by subsequent reads again. To erase the block, it requires reading the existing data on the target and copying all of them to a new block. As shown in Figure 2c, this internal task, called *read reclaiming* [69], can deteriorate the read performance seriously. In this test, we intensively read a set of specific blocks four times as a precondition and read them again in a random I/O pattern. One can observe from the figure that the read latency affected by the read reclaiming reaches as high as 2.5 ms, which is $32\times$ longer than the typical cases.

Even with the ideal situation that only utilizes the underlying SSDs as read-dedicated storage, this long-tail latency imposed by the read reclaiming are inevitable, and thus, it is necessary to devise new interface and firmware assistance to get them off the critical path in LSM KV’s read services.

3 Motivation and Related Work

3.1 Long-tail Latency on Reads

Figure 3a shows the cumulative distribution function (CDF) of Get latency for diverse RocksDB usage scenarios of Facebook [12, 73] and Yahoo [74]. In this evaluation, we use RocksDB 6.23 [1] on a baseline 1.92TB NVMe hardware that we will modify in Section 4.1 and use for all the remaining tests. This baseline employs 3GB internal DRAM and includes 64 TLC NAND flash (64 layers), which are connected to eight different channels. The detailed environment descriptions are the same as what we used for Section 7.

Significance of long-tail latency. Thanks to the low device-level latency of flash, the nominal performance trend of them is similar to each other; all their Get latencies are under 200 μ s. However, the Get latencies reach a few ms from three nine

(P99, 99.9th percental), and all their latencies increase compared to the normal Get latency as high as $15.7\times$. The main reason why this long-tail latency is observed across all the RocksDB usage scenarios that we test does not stem from database or kernel computation but heavy storage accesses. To be precise, we also decompose the execution time of UserDB, Facebook’s social graph data processing workload [12, 73], into Get’s storage latency (*Storage*), client computation times (*App*), database latency (*RocksDB*), and kernel latencies (*Kernel*). As shown in Figure 3b, the computation of LSM KV’s software stack does not sit on a critical path in the Get long-tail latency, but *Storage* takes 87% of the total execution time thereby dominating Get service times at the tails. While the computation of latency of software stack is well balanced with *Storage* (taking half of the total execution time), LSM KV’s heavy I/O requests sharply increase the fraction of *Storage* when it should reclaim Memtables and/or SSTFiles.

Internal tasks’ performance impacts. Figure 3c shows a time series analysis for the UserDB workload and compares its read characteristic with an ideal Get-only workload, which exhibits I/O patterns the same as UserDB but removes all Put queries from its execution. One can observe from this analysis that, when RocksDB flushes Memtable (at 596K index), the baseline read latency increases from $147\mu s$ to $2.97ms$, and the read latency does not return for a while. Similarly, once RocksDB begins to compact SSTFiles, it introduces many reads and writes to merge KV pairs, which unfortunately block incoming Get requests thereby exhibiting $30\times$ longer latency than the normal cases. Note that the latency of reads being performed in parallel with WAL is not that significant (compared to flush and compaction), but WAL also makes the Get latency $10.6\times$ worse than the usual cases.

3.2 Scheduling Internal Tasks

Challenges of system-level approaches. There are many studies [4, 5, 7–10, 15, 16, 18–20, 22–24] that try to address the performance degradation imposed by RocksDB’s internal tasks, such as TRIAD [22], PebblesDB [23], and SILK [24]. There are variant optimization points across these approaches, but their proposals in general reschedule or delay flush and compaction into idle or other available times, thereby removing the long-tail latency. While these system-level approaches can hide the read/write overhead imposed by LSM KV’s internal tasks to some extent, they cannot remove the long-tail latency on Get services because of unavailability to handle SSD’s internal tasks and side-effects raised by their scheduling. Specifically, postponing the compaction removes the suspending time for incoming Put services, but it enforces LSM KV’s L0 accumulatively accommodate SSTFiles without a data migration to L1, thereby increasing the Get latency. Figure 4a compares two tail latency trends on Gets, each being served with and without compaction rescheduling. The

Get tail latency is sustainably managed when RocksDB compacts SSTFiles at the right time (lower than $1ms$), but its tail latency served with the delayed compaction keeps increasing and reaches $3.1ms$, which is $3.6\times$ longer than the no-scheduling case of RocksDB compactions. This is because Get services require searching the appropriate values (paired with input keys) from the beginning to the end of RocksDB’s L0. Since the SSTFiles on L0 are not sorted, the KV searching introduces many outstanding reads thereby increasing the tail latency.

On the other hand, as shown in Figure 4b, the delayed flush of RocksDB also increases the Get tail latency as high as $27.4\times$. The reason why the Get tail latency looks more severe than the rescheduled compaction is that delaying Memtable flush gobbles up all the in-memory spaces, allocated to Memtable management. Thus, the writes of RocksDB are all stalled until it secures a Memtable, which in turn makes the read service suspended seriously.

Device-level latency determinism and limits. The aforementioned SSD’s internal tasks are well-known challenges to exhibit serious performance drop and long latency [31–44, 75–77]. Since the internal tasks are invoked in an arbitrary time period, they render many productions in diverse computing domains difficult to deploy latency-critical applications in the environment. Recently, the standard NVMe protocol introduces the *predictable latency mode* (PLM) interface in an attempt to make the latency predictable and deterministic. PLM proposes that SSDs operate in either a deterministic performance window (*DTWIN*) or a non-deterministic performance window (*NDWIN*). Based on the NVMe specification [45], *NDWIN* is the time period to prepare the next *DTWIN*.

Note that PLM interface is simply a part of interface protocol, which does not enforce specific requirements or design details for the guarantee of deterministic latency. While this young interface presents blueprints on how to handle the unpredictable SSD behaviors in a well-managed manner, *PLM is in practice a just best-effort contract, which only supports soft latency determinism*. For example, we cannot make the underlying SSD always appropriately work with *DTWIN* because SSD’s internal tasks for hiding the flash characteristics are inevitable to invoke. Even in the ideal case where the underlying hardware hides all the SSD’s internal tasks with its maximum efforts, the latency determinism can be easily broken according to how the host-side LSM KV behaves at anytime. To support strong latency determinism, it is necessary to have a close collaboration between the host-side LSM KVs and the underlying storage.

4 High-level View of Vigil-KV

The main goal of this work is to secure an LSM KV system that has no long-tail latency on Get services to make their read performance deterministic and consistent. As this strong latency determinism is infeasible to achieve by scheduling

either only LSM KV's or SSD's internal tasks alone, Vigil-KV takes a hardware and software co-design approach. Specifically, Vigil-KV hardware is designed towards offering basic scheduling blocks that allow the host to integrate strong latency determinism into the LSM KV. On the other hand, the software part of Vigil-KV classifies the requests of LSM KV's at runtime and carefully assigns them to appropriate by fully utilizing the scheduling blocks that the underlying hardware provides. This hardware and software co-design approach can make the latency of client-side Get queries consistently deterministic and have no long-tail all the time.

4.1 Hardware Support for Fine-Granular Performance Windows

PLM interfaces. As shown in Table 1, Vigil-KV hardware implements and provides a set of PLM interfaces that allow the host-side Vigil-KV software to precisely schedule the device states. The functionalities that our PLM interfaces offer are largely classified into three: i) PLM setup (`PLMConfig()`), ii) NDWIN and DTWIN configuration (`PLMWindow()`), and iii) device log queries (`GetLogPage()`). The table also includes how the host-side kernel driver can implement those three semantics using NVMe feature commands. For example, a LSM KV system's kernel driver can turn on or off the target storage's PLM mode by configuring a feature ID (PLM configuration) and enable flag (on/off) through NVMe's `set-feature` [45]. In similar way, it can simply configure the performance window of Vigil-KV hardware using `PLMWindow()`. To query the device state/condition information (that we will reveal in Section 5.2), the LSM KV system can communicate with Vigil-KV hardware through `GetLogPage()` simply returning the results into 512B data package, called a *log page*. Based on a given performance window information, Vigil-KV hardware prioritizes NDWIN to perform SSD's internal tasks as much as possible, and it guarantees that the internal tasks are not scheduled in DTWIN. As discussed in Section 3.2, SSD's internal tasks cannot permanently be postponed, we regulate the longest-serving time of DTWIN and reports it to the host through `GetLogPage`. In addition, Vigil-KV hardware defines the minimum time of NDWIN that should be secured to handle SSD's internal tasks and exposes the configuration time to the host via the log page. Thus, Vigil-KV software can utilize this information by referring into the log page to schedule performance windows appropriately.

NVM multi-set architecture. To offer a variety of performance scheduling options to the host, our hardware also introduces *NVM multi-set*, which splits the backend storage into multiple volumes, each being exposed to the host as a separate PCIe physical function. Vigil-KV hardware then enables the PLM interface to each physical function, called *NVM set* and makes them work independently by allocating different internal logic/cores across the sets. This NVM multi-set archi-

PLM semantics	NVMe cmd	Field name		
		OP CODE	Feature ID (CDW10)	NVM Set ID (CDW11) Feature Enable (CDW12)
<code>PLMConfig()</code>	Set	PLM Config	SetID	Enable (0: Off, 1: On)
Arg1: SetID	Features			
Arg2: Enable				
<code>PLMWindow()</code>	Set	PLM Window	SetID	Enable (0: NDWIN, 1: DTWIN)
Arg1: SetID	Features			
Arg2: Enable				
<code>GetLogPage()</code>	Get	Return values		
Arg1: SetID	Features	Longest-serving time of DTWIN, Preserved NDWIN, Device lockdown		

Table 1: Vigil-KV hardware.

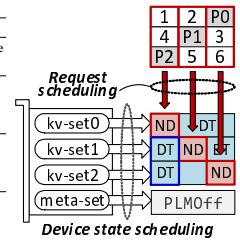


Figure 5: Vigil-KV software.

ture can grant maximum flexibility to the host-side LSM KV's software components, such that they schedule the underlying device states (DTWIN and NDWIN) in a finer granular manner. For example, the LSM KV system can configure different performance windows within a single NVMe device by configuring the NVM Set ID of NVMe's `set-feature` (i.e., the codeword 11 of NVMe's command) differently.

4.2 Software-Defined Strong Latency Determinism for Get services.

Figure 5 shows how Vigil-KV software achieves strong latency determinism by utilizing the finer-granular performance windows that Vigil-KV hardware provides. It consists of three major logical components: i) *metadata separation*, ii) *device state scheduling*, and iii) *request scheduling*.

Managing data, devices, and requests. Vigil-KV software excludes a physical function from the storage volume and internally allocates it for metadata management, called *meta-set*. PLM of this meta-set is disabled by `PLMConfig()`, and Vigil-KV software isolates all WAL and journaling activities from LSM KV's regular queries by forwarding the metadata-related requests to the meta-set. This metadata separation allows the kv-sets not to be interfered with by the heavy internal writes for crash consistency management, such that our device state and request scheduling mechanisms can mainly focus on offering strong latency determinisms for Get services.

On the other hand, the remaining physical functions that Vigil-KV hardware exposes are allocated to handle incoming LSM KV's query requests at the kernel-level, which is referred to as *kv-sets*. Vigil-KV software then schedules all the kv-sets device states (i.e., performance windows) to make $n - 1$ kv-sets be in DTWIN at any given time (using `PLMWindow()`) while allowing NDWIN to be granted to the underlying kv-sets in a fairly scheduled aspect (round-robin). n is the total number of physical functions that Vigil-KV can assign to the SSTFile management. Vigil-KV software classifies LSM KV's internal tasks and client requests at runtime and schedules them differently by knowing the underlying device's configured performance windows. Specifically, all the client requests are scheduled to be served from the $n - 1$ kv-sets, configured with DTWIN. In contrast, Vigil-KV software

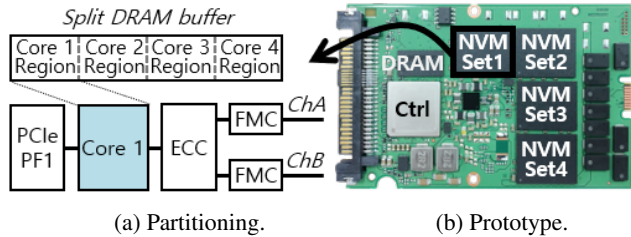


Figure 6: Vigil-KV hardware prototype.

schedules all the requests coming from LSM KV’s internal tasks with a kv-set, scheduled by NDWIN (if there is), but regulates the number of the internal tasks’ requests not to make NDWIN be too much long, thereby having always $n - 1$ kv-sets configured with DTWIN. We will explain the details of this device state and request scheduling in Section 6.2.

Data reconstruction for NDWIN. Vigil-KV pushes all the LSM KV’s and SSD’s internal tasks into NDWIN, which are scheduled across different kv-sets in a round-robin manner. While handling requests over NDWIN is essential for both the LSM KV and SSD, the client requests, particularly Get services, targeting the kv-set scheduled with NDWIN can be blocked, thereby exhibiting the long-tail latency. To address this, Vigil-KV encodes parity bits and writes them with internal tasks at NDWIN. Specifically, when Vigil-KV stores an SSTFile, it splits the file into multiple chunks and stripes those chunks across kv-sets at NDWIN. Since we ensure that there are $n - 1$ kv-sets configured DTWIN at any given moment, Vigil-KV reads out the data from other kv-sets, reconstructs the original data, and serves them without touching the NDWIN kv-set. This data reconstruction inspired by emerging “array-level” memory and storage techniques [48, 78–81] can obviously remove the long tail latency on Get services, but its reconstruction time can increase the average latency compared to ideal storage making all kv-sets DTWIN consistently. Thus, Vigil-KV also minimizes NDWIN to avoid unnecessary data reconstruction at the physical function level. Note that, as the parity bits are generated per chunk (not per SSTFile) in our scheme, it does not need to recalculate the parities after compaction. The details of this technique and implementation will be described in Section 6.2.

5 Hardware Prototype and Characterizations

5.1 Enabling PLM with NVM Multi-Sets

Partitioning an SSD. Modern SSDs employ many flash packages, which are connected to multiple embedded cores through multiple memory buses, called *channels*. All flash packages per channel are managed by a specific micro-coded controller, called flash memory controller (*FMC*). For example, our baseline hardware (SSD) contains eight flash packages, each containing eight flash memory banks, and all of them are connected to four cores through eight channels and FMCs. Since each FMC manages the underlying flash pack-

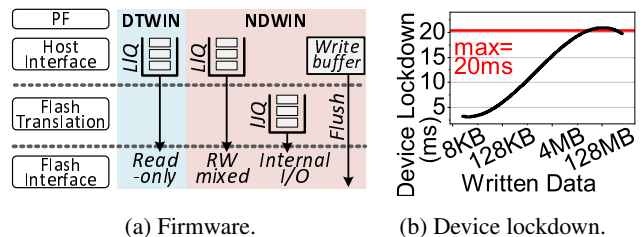


Figure 7: PLM and its constraint.

ages in a self-governing manner, we modify the baseline hardware to partition the single storage space into multiple spaces. Specifically, as shown in Figure 6a, we allocate each core to every two FMCs and make all the cores work independently as a (separate) physical function. As each physical function should not interfere with each other, we also evenly split the internal DRAM space into multiple spaces, each being allocated to a different physical function. Figure 6b shows our prototype of Vigil-KV hardware. There are four physical functions, each being able to be indicated by a different identifier from the host (cf. Table 1’s NVMSet ID). Flash firmware is instantiated per core, such that a physical function performance is not interfered with by other physical functions.

Integrating PLM. To implement DTWIN, each firmware of Vigil-KV hardware employs multiple queues, each being associated with the host command control and internal task management (Figure 7a). Specifically, the internal job queue (*IJQ*) is dedicated to a firmware module that manages address translation while legacy *I/O* queues (*LIQ*) are allocated to the firmware part that manages the host (NVMe) interface. The requests in Vigil-KV hardware can be therefore classified into legacy and internal tasks and served differently using *IJQ* and *LIQ*. Specifically, if a physical function is configured with DTWIN (using `PLMWindow()`), our firmware only handles the requests coming from *LIQ* and suspends all the requests of *IJQ* in both foreground and background. This device can immediately serve the incoming (client) read requests without an interruption of SSD’s internal tasks. However, the firmware cannot suspend the requests of *IJQ* if there is no room, which enforces the host schedule DTWIN appropriately. We will explain this constraint in detail shortly.

5.2 PLM Constraint and Behavior Analysis

DTWIN/NDWIN conditions. While resource partitioning and queue isolation (*IJQ/LIQ*) can remove the read latency spikes imposed by SSD’s internal tasks, unfortunately, making deterministic latency consistent is not that simple; it needs a strong collaboration with the host. First, read services on DTWIN suffer interference from a write, which was buffered in a previous NDWIN state. To eliminate this interference, Vigil-KV’s firmware explicitly flushes the internal buffer before jumping into DTWIN and disables the buffer for further writes in DTWIN. Note that offering DTWIN with fewer restrictions is the mission that our hardware targets to

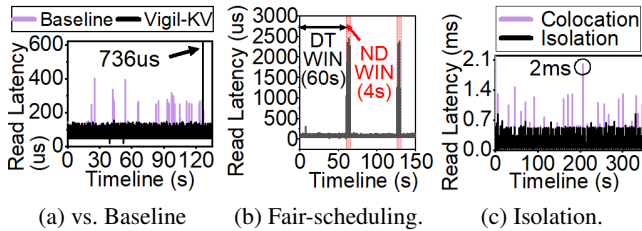


Figure 8: Performance characterization of prototype.

achieve, but it should not lose any data during I/O services with DTWIN. Thus, it is necessary to clear the internal buffer and bypass it before DTWIN and in the middle of DTWIN, respectively. During the internal buffer flush, the host should not further write data in order to clearly wipe it out, which is called *device lockdown* condition. Figure 7b analyzes the device lockdown times varying based on how much data were written in the previous NDWIN. All the workloads that we tested [12, 74] write tens of MB during NDWIN. it is sufficient for the host to hold the data (if there is) by under 20 ms. Similarly, when there is a write on DTWIN, our hardware returns the performance window from DTWIN to NDWIN in order to guarantee strong durability and consistency of the written data. The host therefore makes sure that there is no write on DTWIN, called DTWIN’s *write-free* condition.

DTWIN must also not hurt the current level of reliability management that the existing flash firmware provides. Specifically, the underlying flash media can be stressed only with reads even though there is no write or internal task because of the read disturbance issue (Section 2.2). Thus, Vigil-KV hardware regulates the most extended time window for DTWIN, called *maxDTWIN*, by considering the worst case where the heavy reads on a specific block can corrupt all the page data therein. Similarly, NDWIN should be continued for a certain level of the time duration, called *minNDWIN*, which is the shortest time to complete SSD’s internal tasks (data migration and block erases) and the accumulated requests in IJQ during *maxDTWIN*. Obviously, these *maxDTWIN* and *minNDWIN* periods are strongly correlated because IJQ is limited to queue SSD’s internal tasks. By considering this, the host should schedule DTWIN and NDWIN fairly, called *fair-scheduling* condition. Based on preliminary profiles, we configure *maxDTWIN* and *minNDWIN* as 60 and 4 seconds, respectively.

Note that all these information such as the device lockdown time, *maxDTWIN*, and *minNDWIN* are exposed to the host through `GetLogPage()` (cf. Table 1).

Performance characterization and validation. Figure 8a compares the read latency trends between the baseline device and our Vigil-KV hardware prototype. While the baseline device exhibits multiple latency spikes ($\sim 402us$), the reads with Vigil-KV hardware are all served by $74us$, on average, and it is guaranteed for the latency to be under $200us$. Note that, when we change DTWIN to NDWIN by calling `PLMwindow()` at 128 seconds, the read latency reaches as high as 736

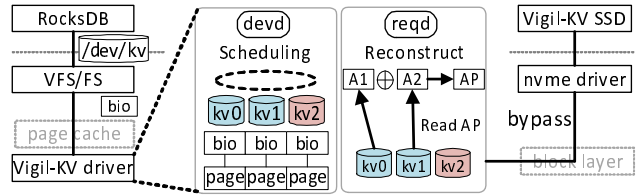


Figure 9: Implementation of Vigil-KV software stack.

us as SSD’s internal tasks are scheduled in that performance window correctly. When we schedule DTWIN and NDWIN one by one (by satisfying the fair-scheduling), as shown in Figure 8b, the performance behaviors mentioned above are all guaranteed across multiple DTWINs. At the same time, the hardware is busy handling the accumulated internal tasks in NDWIN. Lastly, Figure 8c compares the baseline device that collocates reads and writes within a single storage space and Vigil-KV hardware isolating the interference across multiple physical functions. One can observe from this figure that the read latency of the baseline device severely fluctuates and reaches as high as 2 ms. In contrast, the read latency on a physical function of Vigil-KV hardware is not interfered with by the writes heading to other physical functions even though we turned off the PLM interface for the physical function. This is because we partition each physical function with completely different resources. Note that Vigil-KV software utilizes this performance isolation for metadata management, which can make kv-sets free from managing the write-free condition in cases of writing WAL and journaling to the underlying storage.

6 Details of Vigil-KV Software

While there are constraints for PLM management, Vigil-KV hardware opens the opportunity to schedule performance windows across different physical functions being mapped to NVM sets in a finer granule manner. Vigil-KV software separates LSM KV’s internal tasks, including metadata management from client Get services, and schedules them with NDWIN having SSD’s internal tasks together. In addition, the software part of Vigil-KV reconstructs data in cases where it cannot serve the data because NDWIN services, which can in turn allow LSM KVs to have DTWIN consistently, providing strong latency determinism.

6.1 Vigil-KV Stack Implementation

Figure 9 shows the implementation of our Vigil-KV software stack. RocksDB connects Vigil-KV hardware through existing file system interfaces and performs SSTFile-related services on `/dev/kv`. Underneath the file systems, we locate our Vigil-KV driver operating with two kernel threads, `reqd` and `devd`, each scheduling block I/O (`bio`) requests and our hardware’s device states, respectively. Vigil-KV driver maps

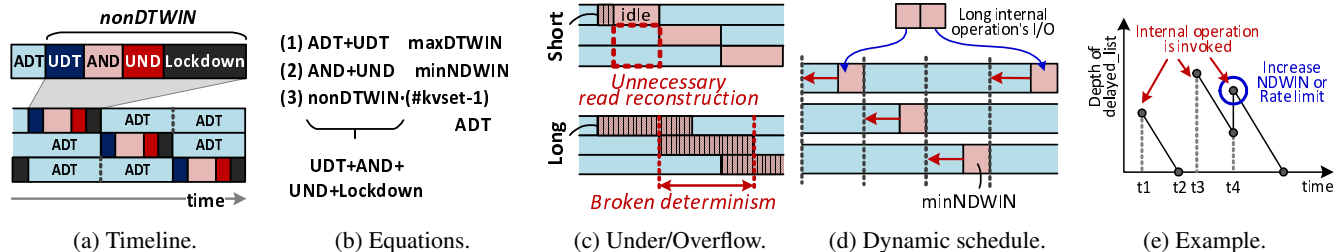


Figure 10: Performance window management.

multiple physical functions that Vigil-KV hardware exposes to different NVM sets (meta-set and kv-sets) at the system’s initialization. `reqd` is similar to Linux existing multiple device `md` driver for striping data chunks and bios across different kv-sets, but it schedules them being aware of underlying device states. Specifically, `reqd` ensures the scheduled bio requests satisfy the write-free condition on DTWIN. In addition, it makes sure that the client’s read requests are not stalled due to LSM KV’s internal tasks by performing the data reconstruction on-the-fly. On the other hand, `devd` schedules the device states for kv-sets by considering the fair-scheduling condition and device lockdown time. More details of this device state scheduling will be explained shortly.

To make read latency predictable, the Vigil-KV driver also bypasses Linux page cache and block layer, which can make the read latency fluctuate and/or be difficult to manage to some extent. For example, Since kv-sets are only managed internally, the bio structures for kv-sets (e.g., logical block address and offsets) are different from the bio requests that the page cache manages. Instead, the Vigil-KV driver employs an internal buffer, called `plm_cache`, which buffers bio requests of kv-sets in a form of Linux `stripe` list. The `plm_cache` size can be configured by the user as a kernel parameter at the boot time. When `reqd` schedules the including bio requests to underlying kv-sets, it thus uses `stripe` requests. Since the Vigil-KV driver bypasses the page cache, it also offers `plm_sync` system call (a variant of file system’s `fsync`) to RocksDB. This `plm_sync` makes sure that the Vigil-KV driver completely flushes `plm_cache` before Memtables, WAL, and SSTFiles are deleted because of LSM KV’s internal tasks. The reason why our Vigil-KV driver bypasses the block layer and directly communicates with the `nvme` driver is that the block layer’s bio merging and ordering can break determinism. For example, the requests of LSM KV’s internal tasks are scheduled for NDWIN, but they can be issued at DTWIN by the block layer. Note that as `reqd` schedules LSM KV’s internal tasks and client requests differently, it is required to deliver the priority information from LSM KV to the Vigil-KV driver. To this end, we have a minor modification on RocksDB and journaling block device daemon (`jbd2`), which can be easily applied to other LSM KVs. When RocksDB creates background threads, it calls a system call, `ioprio_set` that configures I/O priority as ‘internal task’. `ioprio_set` delivers the priority by storing its information into `io_context` of process control block,

`task_struct`. Since Get queries and WAL are managed by all the same thread of RocksDB, we modify `WriteImpl()` such that it configures I/O priority as ‘WAL’ by calling `ioprio_set` before performing `WriteToWAL()`. Journaling is also classified by `ioprio_set` before committing a transaction (e.g., `jbd2_log_do_checkpoint()`).

6.2 Performance Window Management

Device state scheduling. As shown in Figure 10a, `devd` schedules DTWIN and NDWIN to make sure that there are always $n - 1$ kv-sets, configured with DTWIN. Therefore, all the client requests are served from DTWIN or `reqd`’s data reconstruction. When `devd` schedules performance windows to meet the fair-scheduling and device downtime constraints, there are two more technical challenges. Even though `reqd` reads the data from kv-sets configured with DTWIN, the read request can be delayed because of the outstanding reads issued previously and not yet completed. These delayed reads can be served at NDWIN, which cannot in turn offer the strong latency determinism. Similarly, the writes issued to NDWIN can be practically served at DTWIN because of the outstanding writes as well as the time delay caused by SSD’s internal tasks to some extent. This situation is less desirable than the former as it can break the write-free condition on DTWIN.

We classify DTWIN and NDWIN more specifically by considering those two unavailabilities further. DTWIN is split into *ADT* (available DTWIN) and *UDT* (unavailable DTWIN), and similarly, NDWIN is also separate into available/unavailable NDWIN (*AND/UND*). Since UDT and UND can have such outstanding operations on `reqd`, `devd` schedules NDWIN and DTWIN with a time unit as long as *nonDTWIN*. *nonDTWIN* includes UDT, AND, UND, and the device lockdown time windows (*Lockdown*), and each of the windows should satisfy the condition described by Equations 10b. `devd` profiles kv-sets’ bandwidth and then estimates UDT and UND by dividing the total amount of data volume for the outstanding requests with the read and write bandwidth of kv-sets, respectively. Note that, in contrast to the read bandwidth, the write bandwidth on DTWIN can vary. We thus use the worst-case bandwidth of writes for the UND estimation.

Dynamic adjustment for NDWIN. When `devd` controls performance windows of the underlying kv-sets per *nonDTWIN*

(=ADT), there are two challenges that it needs to address as shown in Figure 10c: NDWIN *underflow* and *overflow*. In cases where the amount of internal tasks of LSM KV and/or SSD at NDWIN, reqd wastes computation for data reconstruction and can increase the read latency when there is heavy read traffic; we observed that, when one increases QPS (queries per second) from 60K to 150K, the read latency increases by 26%, on average. While minimizing the data reconstruction involvement (NDWIN) is the matter, NDWIN can break determinism if too many the internal tasks are issued. We also preliminary evaluated all our workloads and observed that 48.3% of compaction scheduled at NDWIN (excluding UDT, UND, and Lockdown) are executed at DTWIN.

As shown in Figure 10d, devd adjusts NDWIN at runtime to address the underflow and overflow situation. Specifically, devd begins scheduling NDWIN (per ADT) by setting it as long as minNDWIN to minimize the involvement of reqd’s data reconstruction and spreads buffered bio requests (stored on plm_cache) across different minNDWIN. If the outstanding requests are accumulated more than a threshold, it maximizes NDWIN to serve LSM KV internal tasks’ I/O as fast as possible. For example, as shown in Figure 10e, devd examines reqd’ queues containing outstanding bio requests at the time epoch 1 ($t1$). In this case, all the outstanding requests are served/completed before $t2$. However, since the requests associated with LSM KV internal tasks at $t3$ are not resolved by $t4$, devd maximizes NDWIN. Note that, Vigil-KV applies dynamic NDWIN adjustment for plm_cache by using the user-defined memory limits as adjustment threshold.

7 Evaluation

7.1 Experimental Setup

Prototype and environments. We implement a prototype of Vigil-KV hardware on a 1.92TB datacenter-scale NVMe SSD for research purposes. Vigil-KV hardware employs four physical functions, and they equally divide the hardware resources such as 3GB LPDDR4 DRAM, eight channels, and 64 TLC NAND flash dies into four. Note that the baseline hardware is not that different from the Vigil-KV hardware. It has hardware resources the same as Vigil-KV hardware, but

	Short description.	Get (%)	Put (%)	Get \$ hit (%)	Flush write (GB)	Compaction write (GB)
FB	UserDB All social graph actions	54	46	25	2.4	8.8
	ZippyDB Read ObjStorage meta	42	58	86	8.9	17.5
YCSB	A Log user action	66	34	17	3.6	19.2
	B Update/read photo tag	95	5	23	0.2	1.3
	D Read latest record	95	5	83	0.5	1.8
	F Update user record	74	26	45	3.1	15.6

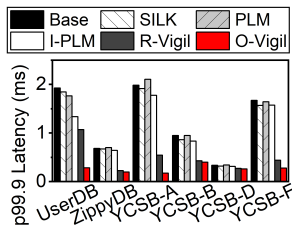
Table 2: Important characteristics of evaluated workloads.

it only employs a single physical function. We perform the evaluation on a 12-core AMD Ryzen 9 5900X, 96GB DRAM, and Vigil-KV hardware by running Vigil-KV software implemented on RocksDB 6.23.0 and Linux 4.19.91. For the evaluation, we set the size of plm_cache as 2GB.

Workloads. We evaluate six workloads that use an LSM KV as their backend storage engine. (two from Facebook [12] and four from Yahoo [74]) For social network services, Facebook uses UserDB and ZippyDB workloads that serve social graph data and object (e.g., image or video) storage metadata as a form of key-value, respectively. The key-value cache hit ratio of UserDB is only 25% due to its irregular key access pattern, whereas that of ZippyDB is 86% because of its read-latest characteristics of social contents. Yahoo also provides representative LSM KV access patterns of various cloud services, such as write-intensive (YCSB-A), read-intensive (YCSB-B), read-latest (YCSB-D), and read-modify-write intensive (YCSB-F). In the table, also analyze the amount of writes caused by internal tasks (compaction) during our evaluations.

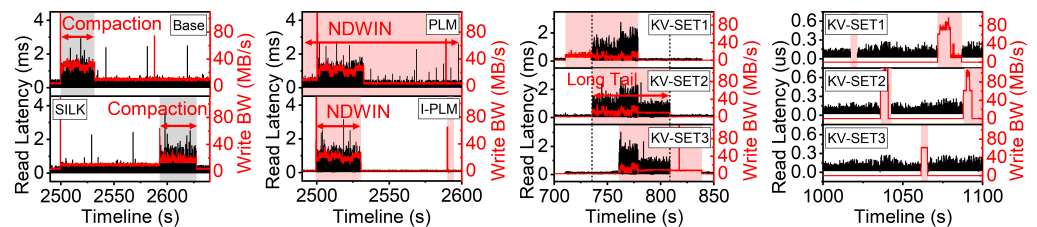
Configurations. We evaluate six different LSM KV hardware and software combinations.

- Base [1]: the representative conventional LVM KV (e.g., RocksDB) with the baseline hardware.
- SILK [24]: the state-of-the-art software supports of LSM KV with the baseline hardware.
- PLM: Vigil-KV hardware with a simple driver, which utilizes the PLM interfaces (cf. Section 4.1).
- I-PLM: based on PLM, we add the metadata isolation support (cf. Section 6.1).
- R-Vigil: based on I-PLM, we add the device state scheduling support (cf. Section 6.2).
- O-Vigil: based on R-Vigil, we add the dynamic non-determinism scheduling support (cf. Section 6.2).



(a) p99.9 tail latency.

Figure 11: Tail latency.



(a) Base and SILK.

(b) PLM and I-PLM.

(c) R-Vigil.

(d) O-Vigil.

Figure 12: Time series analysis of representative workload (UserDB).

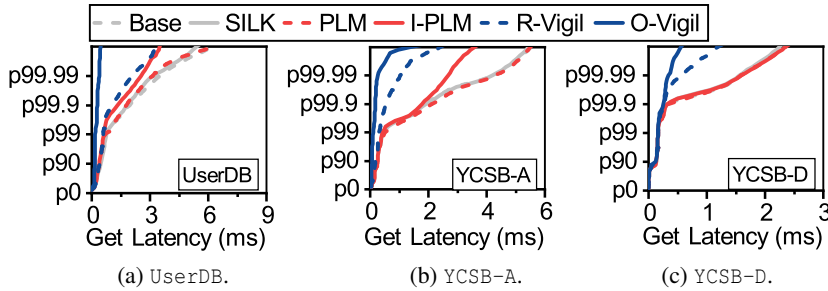


Figure 13: CDF graphs.

Since Vigil-KV adopts the concept of array-level memory and storage techniques making three NVM sets as a single storage volume, we use conventional multiple device (md) driver for Base and SILK as well to satisfy the fair performance comparison among the configurations that we tested.

7.2 Long-tail Latency Analysis

We analyze the long-tail latency on Get services by executing all the Facebook and Yahoo workloads atop the six configurations. As shown in Figure 11, UserDB, YCSB-A, and YCSB-F show longer p99.9 tail latency than others due to their high Put service ratio, which increases the LSM KV and SSD internal tasks. Meanwhile, ZippyDB and YCSB-D further exhibit shorter tail latency than YCSB-B thanks to their high key-value cache hit ratio, which can reduce the number of reads interfered with by internal tasks. To understand how six different configurations impact long-tail latency, Figure 12 analyzes time series by selecting UserDB as a representative workload. It shows both storage read latency (left axis with black line) and write throughput (right axis with red line), which allow us to infer how much LSM KV and SSD internal tasks interfere with Get queries and when the internal tasks occur.

Base vs. SILK. As shown in Figure 11, SILK only reduces 5% of the tail compared to Base, on average. Even though UserDB can capture the user idle behaviors while YCSB workloads cannot, still there was not an enough idle time to schedule LSM KV internal tasks as shown in Figure 12a. Thus, the delayed compaction starts to interfere with Get services from 2600 seconds in SILK, and the storage read latency spikes.

SILK vs. PLM. PLM experiences the tail similar to Base (5.4% longer than SILK), which indicates that Vigil-KV hardware cannot guarantee latency determinism without Vigil-KV software. As shown in the top of Figure 12b, Vigil-KV hardware is in NDWIN most of the time (red background) since WAL or journaling breaks the DTWIN’s write-free condition.

PLM vs. I-PLM. Therefore, I-PLM isolates WAL and journaling to dedicated meta-set and securing NDWIN (white background) as shown in the bottom of Figure 12b. Since UserDB has a higher Put service ratio than others, it experiences 24.4% shorter long-tail latency compared to PLM, while others achieve 12% shorter long-tail latency, on average.

I-PLM vs. R-Vigil. As shown in Figure 12c, R-Vigil

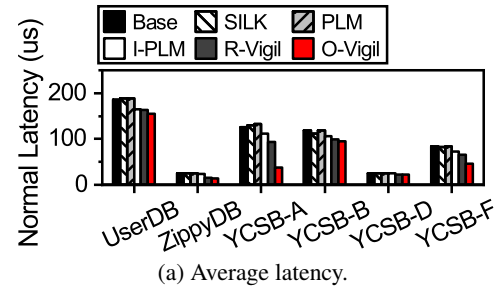


Figure 14: Normal cases.

schedules NDWIN across three kv-sets and guarantees latency determinism as much as possible by reconstructing reads with DTWIN kv-sets. Thus, it can reduce long-tail latency by 48.2% compared to I-PLM, on average. As shown in Figure 11, especially for high Put service workloads (YCSB-A and YCSB-C), they exhibit 70% shorter long-tail latency than I-PLM. However, R-Vigil still exhibits long-tail latency due to NDWIN overflow (cf. 736 ~ 809 seconds in Figure 12c).

R-Vigil vs. O-Vigil. Therefore, O-Vigil strongly guarantees the latency determinism with out dynamic NDWIN adjustment. O-Vigil reduces long-tail latency by 33.5% than R-Vigil, on average, while high Put service workloads (UserDB, YCSB-A and YCSB-F) can achieve shorter long-tail latency 59%. As shown in Figure 12d, O-Vigil dynamically schedules LSM KV’s internal tasks.

Note that O-Vigil not only reduces the long-tail latency of the Base by 3.19 \times , on average, but also guarantees under 500us Get service latency across all the workloads.

7.3 Analysis of Different-level Latency

Tail latency distribution. To understand the impact of Vigil-KV for the different levels of tail-latency, we analyze the CDF of Get latency for three representative workloads, such as UserDB, YCSB-A, and YCSB-D. Since UserDB and YCSB-A are high Put service ratio workloads, the long-tail of Get latency start from p99 as shown in Figures 13a and 13b. On the other hand, as YCSB-D has higher Get service ratio workloads and most of the Get is serviced from the key-value cache, the long-tail of Get latency start from p99.9 as shown in Figure 13c. Thus, for the YCSB-D workload, O-Vigil achieves 78% shorter p99.99 long-tail latency than Base, while only 11% of the long-tail is reduced at p99.9 latency. Not only for YCSB-D, O-Vigil can further mitigates the p99.99 long-tail latency of UserDB and YCSB-A by 69% and 94%, respectively.

Note that Vigil-KV can also guarantee the strong latency determinism (us-scale latency of Get service) more than four nines long-tail latency (p99.99) as shown in Figure 13.

Normal cases. Since Vigil-KV adds more software supports than conventional LSM KV, it is important to analyze the Get service latency of normal cases to check whether Vigil-KV slows down the average Get latency or not. Figure 14 shows the average latency of Get services for all the workloads and

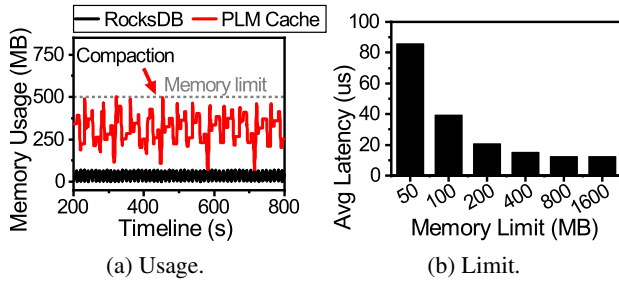


Figure 15: Memory consumption.

configurations, and we observed that Vigil-KV (O-Vigil) reduces the average Get latency by 34% compared to Base (not increases the average Get latency). This is because of two reasons: 1) Vigil-KV isolates the write I/O traffic of metadata (e.g., WAL and filesystem journaling) from the read I/O traffic of Get service, and 2) Vigil-KV minimizes the data reconstruction as much as possible. Note that, the reason why ZippyDB and YCSB-D show shorter average Get latency compared to other workloads is that they have a high key-value cache hit ratio (which is not related to Vigil-KV solutions).

7.4 Memory Consumption and Scan Service

Time series analysis. Figure 15a shows memory consumption of application-level (e.g., RocksDB’s Memtable) and kernel-level (e.g., Vigil-KV driver’s `plm_cache`) during workload execution. We select ZippyDB as a representative workload since there is a large amount of storage writes by compaction internal tasks (cf. Table 2). While RocksDB periodically flushes Memtables to the underlying storage to maintain a certain threshold (e.g., 64MB) of the application-level write buffer, Vigil-KV has to cache/buffer write requests until the target kv-set reaches NDWIN. Thus, the memory usage of `plm_cache` increases when LSM KV internal tasks (e.g., Memtable flush and compaction) occur. However, Vigil-KV supports dynamic adjustment for NDWIN being aware of the memory limits of `plm_cache` which can regulate the maximum memory consumption.

Sensitivity test. To understand the impact of `plm_cache`’s memory limits to the average latency of Get services, we perform sensitivity tests by increasing the memory limits from 50MB to 1.6GB as shown in Figure 15b. While extremely low memory limit (e.g., 50MB) exhibits long normal latency of more than 80 μ s, a few hundreds of MB `plm_cache` is enough to serve normal Get latency without performance degradation.

Note that, although Vigil-KV delays LSM KV internal tasks until target kv-sets reach NDWIN, it does not increase memory footprints or degrade the Get latency, unlike prior studies (e.g., TRIAD [22], PebblesDB [23], and SILK [24]).

Performance with Scan. Vigil-KV mainly considers improving the performance of Get queries as a first-class citizen. This is because Get of most large-scale workloads (Facebook [12] and Yahoo [74]) account for 78% of the total queries in the workloads that we tested. While Scan in contrast accounts

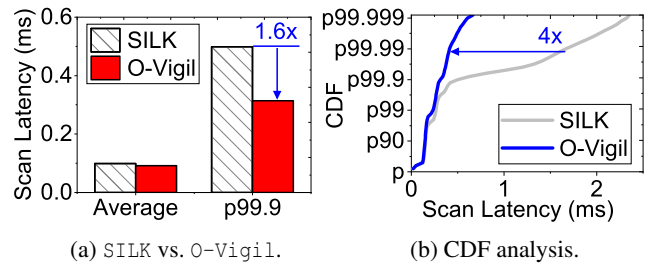


Figure 16: Performance analysis of Scan queries.

for 3% of the total queries, its query latency may also be important for a specific workload such as scanning many posts in parallel (e.g., YCSB-E).

In this subsection, we compare the latency behaviors of Vigil-KV (O-Vigil) with those of SILK by evaluating YCSB-E as the representative of Scan-sensitive workloads (95% and 5% for Scan and Put, respectively). Figure 16a shows the comparison for the average (p50) and p99.9 latency. Since RocksDB performs readahead and prefetch in default, the sequentially of Scan exhibits many cache hits. Similar to YCSB-D, this in turn benefits the average latency marginal as Scan operations of both O-Vigil and SILK from memory than devices in most cases. However, such readahead and prefetch techniques cannot avoid every I/O request and hide all the read latency that the underlying SSD exposes. Vigil-KV can support deterministic latency for such cases, thereby offering 1.6 \times shorter p99.9 latency of SILK. This performance benefit becomes more promising as the degree of the long-tail latency gets higher. As shown in Figure 16b, even though there are few Put operations, they lead LSM KV internal tasks, which can make p99.9~p99.999 tail-latency much longer. Vigil-KV can remove such long-tail latency with hardware/software co-designed strong latency determinism, thereby improving the tail latency by 1.6 \times ~ 4 \times .

8 Conclusion

In this paper, we propose Vigil-KV, a hardware and software co-designed framework that eliminates long-tail latency by introducing strong latency determinism into LSM KVs. We evaluate diverse Facebook and Yahoo scenarios with Vigil-KV, and our empirical evaluation shows that Vigil-KV can reduce the tail latency of the baseline KV system by 3.19 \times while reducing the average latency by 34% as well.

Acknowledgement

The author thanks to anonymous reviewers for their constructive feedback. This work is mainly supported by Samsung (G01200447) and Samsung HiPER. This work is also in part supported by NRF’s 2021R1A2C4001773, IITP’s 2021-0-00524 & 2022-0-00117, KAIST start-up package (G01190015), and KAIST IDEC. Myoungsoo Jung is the corresponding author.

References

- [1] Facebook. Rocksdb: A persistent key-value store for fast storage environments. <https://rocksdb.org>.
- [2] Sanjay Ghemawat and Jeff Dean. Leveldb. <https://github.com/google/leveldb>.
- [3] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The rocksdb experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021.
- [4] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.
- [5] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. Slimdb: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment*, 2017.
- [6] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruva Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, 2017.
- [7] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 2017.
- [8] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. Splinterdb: Closing the bandwidth gap for nvme key-value stores. In *2020 USENIX Annual Technical Conference (USENIXATC 20)*, 2020.
- [9] Yongkun Li, Zhen Liu, Patrick PC Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. Differentiated key-value storage management for balanced i/o performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.
- [10] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.
- [11] Yoshinori Matsunobu, Siying Dong, and Herman Lee. Myrocks: Lsm-tree database storage engine serving facebook’s social graph. *Proceedings of the VLDB Endowment*, 2020.
- [12] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.
- [13] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.
- [14] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)*, 2021.
- [15] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. Slm-db: single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.
- [16] Russell Sears and Raghu Ramakrishnan. blsm: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.
- [17] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.
- [18] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, 2018.
- [19] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference (USENIX-ATC 18)*, 2018.
- [20] Junsu Im, Jinwook Bae, Chanwoo Chung, Sungjin Lee, et al. Pink: High-speed in-storage key-value store with bounded tails. In *2020 USENIX Annual Technical Conference (USENIXATC 20)*, 2020.
- [21] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. Spandb: A fast, cost-effective lsm-tree based kv store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021.

- [22] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. Triad: Creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*, 2017.
- [23] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.
- [24] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.
- [25] Hyojun Kim and Seongjun Ahn. Bplru: A buffer management scheme for improving random writes in flash storage. In *FAST*, 2008.
- [26] Ping Huang, Pradeep Subedi, Xubin He, Shuang He, and Ke Zhou. Flexecc: Partially relaxing ecc of mlcssd for better cache performance. In *2014 USENIX Annual Technical Conference (USENIXATC 14)*, 2014.
- [27] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. Durable write cache in flash memory ssd for relational and nosql databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014.
- [28] Aayush Gupta, Youngjae Kim, and Bhuvan Uргаonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. *Acm Sigplan Notices*, 2009.
- [29] Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee. Fab: Flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics*, 2006.
- [30] Sooyong Kang, Sungmin Park, Hoyoung Jung, Hyoki Shim, and Jaehyuk Cha. Performance trade-offs in using nvram write buffer for flash memory-based storage devices. *IEEE Transactions on Computers*, 2008.
- [31] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiyasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. Fln: Enabling fairness and enhancing performance in modern nvme solid state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018.
- [32] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T Kandemir. Hios: A host interface i/o scheduler for solid state disks. *ACM SIGARCH Computer Architecture News*, 2014.
- [33] Myoungsoo Jung, Wonil Choi, Miryeong Kwon, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut Tayan Kandemir. Design of a host interface logic for gc-free ssds. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.
- [34] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H Noh. Alleviating garbage collection interference through spatial separation in all flash arrays. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*, 2019.
- [35] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 2004.
- [36] Pan Yang, Ni Xue, Yuqi Zhang, Yangxu Zhou, Li Sun, Wenwen Chen, Zhonggang Chen, Wei Xia, Junke Li, and Kihyouon Kwon. Reducing garbage collection overhead in ssd based on workload prediction. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.
- [37] Narges Shahidi, Mahmut T Kandemir, Mohammad Arjomand, Chita R Das, Myoungsoo Jung, and Anand Sivasubramaniam. Exploring the potentials of parallel garbage collection in ssds for enterprise storage systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016.
- [38] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*. Boston, USA, 2008.
- [39] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. *ACM Transactions on Storage (TOS)*, 2017.
- [40] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, 2017.
- [41] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. Errors in flash-memory-based solid-state drives: Analysis, mitigation, and recovery. *arXiv preprint arXiv:1711.11427*, 2017.

- [42] Yu Cai, Yixin Luo, Saugata Ghose, and Onur Mutlu. Read disturb errors in mlc nand flash memory: Characterization, mitigation, and recovery. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2015.
- [43] Bryan S Kim, Jongmoo Choi, and Sang Lyul Min. Design tradeoffs for ssd reliability. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.
- [44] Keonsoo Ha, Jaeyong Jeong, and Jihong Kim. An integrated approach for managing read disturbs in high-density nand flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2015.
- [45] NVM Express, Inc. NVM express specification. <https://nvmexpress.org/specifications>.
- [46] Gyuyoung Park, Miryeong Kwon, Pratyush Mahapatra, Michael Swift, and Myoungsoo Jung. Bibim: A prototype multi-partition aware heterogeneous new memory. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.
- [47] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Nam Sung Kim, Mahmut Taylan Kandemir, and Myoungsoo Jung. Revamping storage class memory with hardware automated memory-over-storage solution. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021.
- [48] Jie Zhang, Gyuyoung Park, David Donofrio, John Shalf, and Myoungsoo Jung. Dram-less: Hardware acceleration of data processing with new memory. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020.
- [49] Huaicheng Li, Martin L Putra, Ronald Shi, Xing Lin, Gregory R Ganger, and Haryadi S Gunawi. Ioda: A host/device co-design for strong predictability contract on modern flash storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.
- [50] Shucheng Wang, Ziyi Lu, Qiang Cao, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. Bcw: Buffer-controlled writes to hdds for ssd-hdd hybrid storage server. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.
- [51] Stan Park and Kai Shen. Fios: a fair, efficient flash i/o scheduler. In *FAST*, 2012.
- [52] Yu Cai, Onur Mutlu, Erich F Haratsch, and Ken Mai. Program interference in mlc nand flash memory: Characterization, modeling, and mitigation. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 2013.
- [53] Xavier Jimenez, David Novo, and Paolo Ienne. Wear unleveling: Improving nand flash lifetime by balancing page endurance. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014.
- [54] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superblock-based flash translation layer for nand flash memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, 2006.
- [55] Bryan S Kim, Hyun Suk Yang, and Sang Lyul Min. Autossd: an autonomic ssd architecture. In *2018 USENIX Annual Technical Conference (USENIXATC 18)*, 2018.
- [56] Myoungsoo Jung, Ramya Prabhakar, and Mahmut T. Kandemir. Taking garbage collection overheads off the critical path in ssds. In *Middleware 2012 - ACM/IFIP/USENIX 13th International Middleware Conference, Montreal, QC, Canada, December 3-7, 2012. Proceedings*, volume 7662 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2012.
- [57] Guanying Wu and Xubin He. Reducing ssd read latency via nand flash program and erase suspension. In *FAST*, 2012.
- [58] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. Exploring system challenges of ultra-low latency solid state drives. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.
- [59] Sungjoon Koh, Junhyeok Jang, Changrim Lee, Miryeong Kwon, Jie Zhang, and Myoungsoo Jung. Faster than flash: An in-depth study of system challenges for emerging ultra-low latency ssds. In *IEEE International Symposium on Workload Characterization, IISWC 2019, Orlando, FL, USA, November 3-5, 2019*. IEEE, 2019.
- [60] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changrim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 477–492. USENIX Association, 2018.

- [61] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 67–80, Santa Clara, CA, February 2016. USENIX Association. ISBN 978-1-931971-28-7. URL <https://www.usenix.org/conference/fast16/technical-sessions/presentation/schroeder>.
- [62] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Error patterns in mlc nand flash memory: Measurement, characterization, and analysis. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012.
- [63] Alaa R Alameldeen, Ilya Wagner, Zeshan Chishti, Wei Wu, Chris Wilkerson, and Shih-Lien Lu. Energy-efficient cache design using variable-strength error-correcting codes. *ACM SIGARCH Computer Architecture News*, 2011.
- [64] Chun-Yi Liu, Jagadish B Kotra, Myoungsoo Jung, Mahmut T Kandemir, and Chita R Das. Soml read: Rethinking the read operation granularity of 3d nand ssds. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.
- [65] Qiao Li, Liang Shi, Chun Jason Xue, Kaijie Wu, Cheng Ji, Qingfeng Zhuge, and Edwin H-M Sha. Access characteristic guided read and write cost regulation for performance improvement on flash memory. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.
- [66] Kai Zhao, Wenzhe Zhao, Hongbin Sun, Xiaodong Zhang, Nanning Zheng, and Tong Zhang. Ldpc-in-ssd: Making advanced error correction codes work effectively in solid state drives. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.
- [67] Hongbin Sun, Wenzhe Zhao, Minjie Lv, Guiqiang Dong, Nanning Zheng, and Tong Zhang. Exploiting intracell bit-error characteristics to improve min-sum ldpc decoding for mlc nand flash-based storage in mobile device. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2016.
- [68] Meng Zhang, Fei Wu, Xubin He, Ping Huang, Shunzhuo Wang, and Changsheng Xie. Real: A retention error aware ldpc decoding scheme to improve nand flash read performance. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2016.
- [69] M Jung and M Kandemir. Revisiting widely-held expectations of ssd and rethinking implications for systems. *SIGMETRICS'13*, 2013.
- [70] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Threshold voltage distribution in mlc nand flash memory: Characterization, analysis, and modeling. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013.
- [71] Neal Mielke, Todd Marquart, Ning Wu, Jeff Kessenich, Hanmant Belgal, Eric Schares, Falgun Trivedi, Evan Goodness, and Leland R Nevill. Bit error rate in nand flash memories. In *2008 IEEE International Reliability Physics Symposium*. IEEE, 2008.
- [72] Yu Cai, Saugata Ghose, Yixin Luo, Ken Mai, Onur Mutlu, and Erich F Haratsch. Vulnerabilities in mlc nand flash memory programming: Experimental analysis, exploits, and mitigation techniques. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017.
- [73] Timothy G Armstrong, Vamsi Ponnekanti, Dhruva Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (MOD)*, 2013.
- [74] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, 2010.
- [75] Wonil Choi, Myoungsoo Jung, Mahmut Kandemir, and Chita Das. Parallelizing garbage collection with i/o to improve flash resource utilization. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018.
- [76] Benny Van Houdt. A mean field model for a class of garbage collection algorithms in flash-based solid state drives. *ACM SIGMETRICS Performance Evaluation Review*, 2013.
- [77] Ming-Chang Yang, Yu-Ming Chang, Che-Wei Tsao, Po-Chun Huang, Yuan-Hao Chang, and Tei-Wei Kuo. Garbage collection and wear leveling for flash memory: Past and future. In *2014 International Conference on Smart Computing*. IEEE, 2014.
- [78] Sangwon Lee, Miryeong Kwon, Gyuyoung Park, and Myoungsoo Jung. Lightpc: hardware and software co-design for energy-efficient full system persistence. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 289–305, 2022.
- [79] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential raid: Rethinking raid for ssd reliability. *ACM Transactions on Storage (TOS)*, 2010.

- [80] Bo Mao, Hong Jiang, Suzhen Wu, Lei Tian, Dan Feng, Jianxi Chen, and Lingfang Zeng. Hpda: A hybrid parity-based disk array for enhanced performance and reliability. *ACM Transactions on Storage (TOS)*, 2012.
- [81] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. Fusionraid: Achieving consistent low latency for commodity ssd arrays. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021.



Pacman: An Efficient Compaction Approach for Log-Structured Key-Value Store on Persistent Memory

Jing Wang[†] Youyou Lu[†] Qing Wang[†] Minhui Xie[†] Keji Huang[‡] Jiwu Shu^{*†}

[†]*Department of Computer Science and Technology, BNRist, Tsinghua University*

[‡]*Huawei Technologies Co., Ltd*

Abstract

Recent persistent memory (PM) key-value (KV) stores adopt the log-structured approach to reap PM's full potential. However, they fail to sustain high performance at high capacity utilization due to inefficient compaction. The inefficiency results from the unawareness of PM's characteristics. This paper proposes PACMAN, an *efficient PM-aware compaction approach* for log-structured KV stores on PM. PACMAN (1) offloads reference search during compaction to service threads, so as to mitigate the onerous index traversal overhead, (2) leverages tagged pointer and DRAM-resident compaction information to avoid excessive PM accesses introduced by garbage collection, (3) redesigns the compaction pipeline based on the PM peculiarities to lower the persistence overhead, and (4) separates hot and cold objects in a lightweight manner to reduce PM data copying in compaction. We apply PACMAN to state-of-the-art PM-based log-structured KV stores and evaluate PACMAN using various benchmarks. Our evaluations show that PACMAN curtails the performance degradation at high capacity utilization, increases the compaction bandwidth by 2-4 \times , and boosts the performance of the state-of-the-art systems by 1.5-1.8 \times under write-intensive workloads.

1 Introduction

The log-structured approach has been adopted in key-value (KV) stores on PM [8, 12, 30, 44] for benefits like high capacity utilization (low fragmentation), small device-level write amplification on PM, and low failure atomicity overhead. Despite having these benefits, the log-structured approach needs to reclaim free space (i.e., garbage collection or compaction) by dedicated background threads (called *cleaners*), which contributes to the major bottleneck, especially under a high capacity utilization [39].

Over-provisioning can alleviate this problem but is not cost-efficient. Datacenters place more emphasis on space utilization in recent years [15] and try hard to fully utilize their storage to reduce the total cost of ownership (TCO) [40]. Thus,

it is important for storage systems to keep high performance under high space utilization, especially for PM that has a much higher cost than traditional storage devices.

The compaction overhead is already severe in DRAM-based log-structured KV stores. For example, RAMCloud's throughput could drop by up to 50% at high capacity utilization [39]. Nibble, a concurrent log-structured KV store, enables 8 compaction threads to carry out the compaction work per socket which has only 15 cores; even so, its throughput drops to nearly a quarter in dynamic workloads [35].

Worse still, we observe that PM's idiosyncrasies exacerbate the bottleneck of compaction. Our evaluation on state-of-the-art PM-based log-structured KV stores¹ (including FlatStore [12] and Viper [8]) shows that their performance drops significantly at high capacity utilization. With abundant CPU resources for compaction (foreground thread count to background cleaner thread count is 3:1), when the capacity utilization increases from 50% to 80%, the system throughput drops by up to 75% under write-intensive workloads. Unfortunately, simply adding more CPU resources for compaction is inefficient, because the performance of PM does not scale well with high thread count [46]; in our experiment, the decline is still up to 60% even with doubling cleaner threads.

We analyze that there are four deficiencies in the conventional compaction approaches of these KV stores accounting for the performance degradation. First, after copying valid objects from the log segment being reclaimed, cleaners need to update object references in the index, which incurs a huge overhead, especially when the index resides on PM due to PM's high access latency (3 \times of DRAM in terms of random read [46]). Second, service threads (i.e., threads performing user requests, such as Get and Put) generate quantities of small random PM accesses for compaction (e.g., marking deleted flags). These small random accesses result in I/O amplification of PM, wasting PM's limited bandwidth (1/3 and 1/6 of DRAM in terms of read and write, respectively [46]). Third, cleaners need to perform many expensive persistence

¹Note that this paper targets PM-based log-structured KV stores but not LSM-tree-based KV stores.

*Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn).

instructions to guarantee crash consistency. Fourth, excessive data copying in compaction contends limited PM bandwidth with service threads. All these deficiencies boil down to *unawareness of PM's characteristics*.

To solve the problems above, this work proposes PACMAN, an efficient PM-aware compaction approach for log-structured KV stores on PM. PACMAN comprises a series of techniques to improve the compaction efficiency of log-structured KV stores on PM. ① PACMAN introduces a technique called *short-cut*, which offloads reference search operations during compaction to service threads. Therefore cleaners can locate and update references without traversing the index. ② PACMAN reduces excessive PM random accesses. Specifically, PACMAN leverages *tagged pointer* to reduce high-latency PM reads, and stores frequently-accessed metadata in DRAM to avoid small random PM writes. ③ PACMAN redesigns the compaction pipeline in a batch pattern and leverages several optimizations according to the characteristics of PM, which accelerates the compaction and reduces the persistence overhead. ④ PACMAN adopts a lightweight hot-cold separation method to reduce the amount of valid data copying on PM and corresponding reference updates. Consequently, PACMAN boosts compaction efficiency, decreases CPU resources for compaction, and enhances system performance at high capacity utilization.

We apply PACMAN to state-of-the-art PM-based log-structured KV stores, FlatStore [12] with different indexes and Viper [8], and evaluate PACMAN using a variety of benchmarks. Our evaluation shows PACMAN enhances the compaction bandwidth by up to 4× and system performance by 2.4-4.6× under write workloads at high capacity utilization. Besides, PACMAN has nearly no side-effects under read-intensive workloads and has little overhead on recovery.

In summary, this paper makes the following contributions:

- We analyze the deficiencies of existing compaction approaches for log-structured KV stores on PM.
- We propose PACMAN, an efficient PM-aware compaction approach for PM-based log-structured KV stores, which enables them to achieve high performance even at high capacity utilization.
- We apply PACMAN to state-of-the-art PM-based log-structured KV stores and conduct a series of experiments to show the efficiency of PACMAN.

2 Background and Motivation

2.1 Log-Structured KV Stores on PM

Benefits of the log-structured approach on PM. The log-structured approach has been adopted by state-of-the-art KV stores on PM [8, 12, 30, 44] for the following benefits. First, a log-structured approach to memory management supports high capacity utilization (i.e., the percentage of space used by alive data) of 80-90%, which is unfeasible for non-copying allocators having great memory fragmentation [39]. Second,

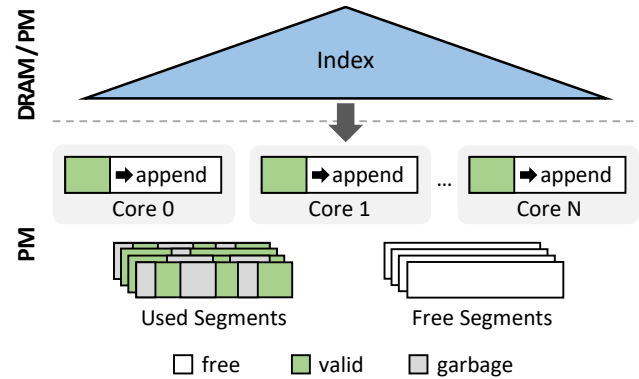


Figure 1: Overview of a log-structured KV store.

due to the mismatch of access granularities between cache lines and PM media (64 bytes vs. 256 bytes of Optane DIMM, the only available PM production for now), small random writes would cause write amplification. The log-structured approach adopts a sequential write pattern, thus alleviating the write amplification and improving the write throughput. Third, in comparison with update-in-place approaches which need expensive logging operations, the log-structured approach makes it easy to commit an arbitrarily-sized persistent write.

Storage structure. Figure 1 shows the basic structure of log-structured KV stores on PM. The whole log space locates on PM and is divided into small-sized (e.g., 4 MB) pieces called *segments*. Each service thread maintains a thread-local segment to append KV objects. Compared to writing to a global log tail, using per-thread segments not only avoids contention but also limits the number of concurrent threads accessing an Optane DIMM [8, 46]. Once a service thread has run out of its local segment, it requests a new free segment from the free segments pool. A global index stores *references* which point to the actual address of KV objects in the log. The index is put in DRAM or PM for different requirements. A volatile index in DRAM delivers better performance but needs more time to restore after a restart. On the contrary, a persistent index in PM provides instant usability after a restart but relatively lower performance.

Garbage collection. Despite having numerous benefits, log-structured systems are obliged to tackle garbage collection by compaction, the main culprit of performance degradation. Update or delete operations make prior objects stale in the log. These stale objects occupy the memory space until being reclaimed. When there is no free space left, service threads stall and wait for new free space produced by compaction. In other words, the system's throughput at high capacity utilization is nearly limited to the compaction throughput.

A qualified compaction process is described as follows. When the fraction of free space is low, the compaction is triggered, and candidate segments are selected to compact through a certain strategy such as cost-benefit score². During

² $score = \frac{(1-u) \times age}{u}$, where u is the segment's utilization (fraction of data alive), and age is the time since the segment running out [39].

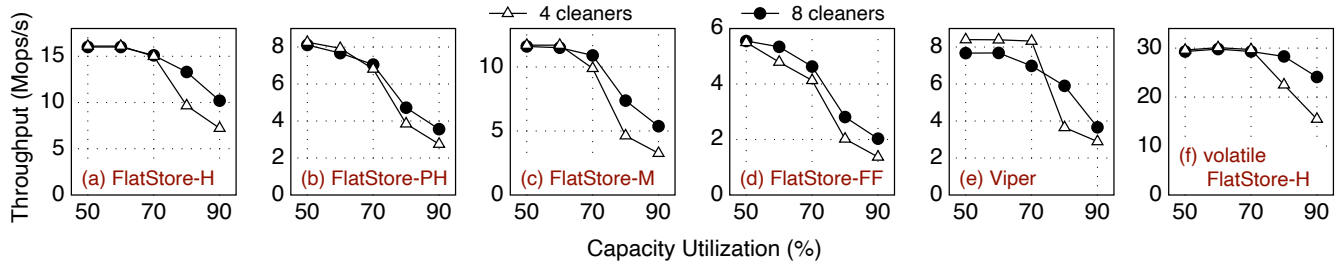


Figure 2: Throughput decline at different capacity utilizations. *FlatStore-H*: with CCEH in DRAM. *FlatStore-PH*: with CCEH in PM. *FlatStore-M*: with Masstree in DRAM. *FlatStore-FF*: with FastFair in PM. *Viper*: with CCEH in DRAM. *volatile FlatStore-H*: both the index (CCEH) and log in DRAM.

compaction, cleaners copy all alive objects from the old segment to a reserved segment and update references to these objects in the index. There are two methods to identify the liveness of each object in the segment. One method is checking whether the object is still pointed by the reference in the index (e.g., FlatStore [12]). Another one is checking the deleted flag (usually 1 bit) in the metadata header of objects; the deleted flag is set when the corresponding object is updated or deleted (e.g., Viper [8]). Both of the two methods have shortcomings on PM, which we will analyze in §2.2. After all valid objects in the old segment have been copied to the reserved segment and their references have been updated, the old segment is cleaned and turned into a free segment.

2.2 Compaction Overhead Analysis on PM

The compaction overhead already matters in DRAM-based log-structured KV stores [35, 39]. Worse still, *the compaction overhead becomes more severe on PM*, as cleaner threads need to contend PM’s limited bandwidth with foreground service threads. Further, necessary but expensive persistence instructions and PM’s high access latency make the overhead of copying objects and updating references higher, especially when the index is persistent.

Experiments. To analyze the compaction overhead of PM-based log-structured KV stores in depth, we evaluate state-of-the-art systems including FlatStore [12]³ and Viper [8]⁴.

We evaluate four versions of FlatStore, including FlatStore-H (FlatStore with CCEH [37], a hash table, in DRAM as a volatile index), FlatStore-PH (FlatStore with CCEH in PM as a persistent index), FlatStore-M (FlatStore with Masstree [34], a trie-like concatenation of B+-trees, in DRAM), and FlatStore-FF (FlatStore with FastFair [20], a B+tree, in PM) to show different cases. Like FlatStore-H, Viper also uses CCEH in DRAM as its volatile index. We measure the performance with a YCSB-A workload, where 200 million KV objects are randomly loaded first, then service threads perform a write-intensive workload (50% Get

and 50% Put) with Zipfian distribution (skewness parameter 0.99) until the system throughput converges to a stable value. The value size is 48 bytes, a representative value of small objects according to recent real-world workloads analysis [9]. We restrict the capacity utilization from 50% to 90% (the percentage of space occupied by alive data). We set the service thread count to 12, and the cleaner thread count to 4 or 8. All threads are bound to a single socket, which is equipped with three Intel Optane DCPMMs.

Figure 2 shows their throughput at different capacity utilizations. From the results we observe that: (1) *The throughput of all systems drops significantly at high capacity utilization, especially for systems with a tree-based or persistent index.* (2) *Simply augmenting more CPU resources for compaction (8 cleaner threads) has limited improvements for log-structured KV stores on PM*, and also intensifies the contentions. Note that the garbage collection overhead will be much larger under uniform workloads; see detailed results in §4.2.3.

We also evaluate the situation that both index and log locate on DRAM (volatile FlatStore-H, Figure 2(f)) to simulate an in-DRAM log-structured KV store. The throughput decline is mitigating than the PM-based KV stores. Using 8 cleaners has obvious improvements on DRAM because DRAM has abundant bandwidth. However, adding more threads on compaction is not cost-effective and has less benefit in PM-based systems as PM does not scale well with multiple threads due to PM’s idiosyncrasies [46]. This shows that PM’s peculiarities aggravate garbage collection overhead.

Overhead analysis. Taking test cases with 4 cleaners and 80% capacity utilization above as examples, we analyze the compaction overheads and find out that there exist four inefficiencies on compaction.

(1) *The high latency of random access in the index.* Lots of random accesses are introduced by two operations on the index, checking references for identifying liveness and updating references after copying alive objects. These two operations require multiple random accesses in the index and these accesses have no cache locality since alive objects in compaction are usually cold.

Viper sidesteps checking references with deleted flags in PM; yet, updating references costs half of the compaction

³As the original FlatStore is a networked system, we implement it as an embedded KV store, and remove the network-related features for simplicity.

⁴We use the variable-sized object version of Viper and enhance it with multi-threaded compaction.

time. FlatStore identifies the liveness of each object by checking the reference instead of using the deleted flag to avoid small random writes. Thus, cleaners in FlatStore need to access the index twice for each alive object, one for checking liveness and one for updating the reference. FlatStore spends 60% of compaction time on the index in FlatStore-H. The overhead becomes more severe when it comes to tree-based or persistent indexes, which reach about 80% and 90% of compaction time in FlatStore-M and FlatStore-FF, respectively. The latency of a search or an update operation in FastFair reaches several microseconds due to PM's high latency.

(2) **Excessive small random access on PM.** Service threads perform excessive PM accesses for garbage information. To maintain the size of garbage data of each candidate segment, service threads read the metadata of the stale object to get its size when the object is updated or deleted. These reads on PM not only incur high latency but also pollute cache. In addition, marking deleted flags in segments would introduce extra small random PM writes. Though marking deleted flags facilitate garbage collection, it is harmful to limited PM bandwidth.

(3) **Expensive persistence instructions.** Though copying alive objects conducts sequential reads and writes, it still costs heavier than copying on DRAM, not only due to the low bandwidth but also because of necessary but expensive persistence instructions (i.e., `flush` and `fence`).

(4) **A large amount of data copying on PM.** The performance slowdown presents superlinear scaling with capacity utilization. Cleaners have to do compaction much promptly at high utilization, and segments to compact have less time to accumulate stale objects. Accordingly, at high capacity utilization, cleaners have to copy bulk of data in segments to reclaim free space. The large amount of data copying contends the limited PM bandwidth with service threads.

To summarize, traditional compaction approaches do not consider the peculiarities of PM, hence squandering limited PM bandwidth. On the other hand, existing log-structured KV stores completely decouple the index and log. Therefore, there has little room to facilitate garbage collection without particular assistance from the index.

3 Design

Motivated by the analysis above, we propose PACMAN, a PM-aware compaction approach for log-structured KV stores on PM. PACMAN solves the deficiencies in conventional compaction approaches according to the characteristics of PM, boosting the system performance at high capacity utilization. PACMAN introduces several core design principles to realize efficient compaction.

- **Avoid onerous index traversal.** PACMAN offloads reference search operations during compaction to foreground service threads without extra effort. With the information offered by service threads, cleaner threads can locate and update references effortlessly (§3.1).

- **Reduce excessive small PM accesses.** PACMAN removes avoidable PM accesses by leveraging tagged pointer and storing frequently-accessed metadata in DRAM (§3.2).
- **Redesign the compaction pipeline to cater to peculiarities of PM.** PACMAN divides the compaction pipeline into two major phases, copying valid objects and updating references. For each phase, cleaner threads process objects in a batched manner. Subsequently, PACMAN can reduce the number of persistence orderings and leverage non-temporal stores and software prefetching (§3.3).
- **Separate hot and cold objects to reduce excessive data copying on PM.** PACMAN uses a hotspot set to distinguish hot and cold objects and stores them in different segments to facilitate compaction. PACMAN can also replace the set silently to handle hotspot shift without blocking foreground service threads (§3.4).

3.1 Traversing Index with Shortcut

Traversing the index has expensive overhead due to the random access pattern, especially for persistent index. PACMAN introduces a technique called `shortcut` to alleviate this overhead in compaction. In this section, we take tree-based (including trie-based) indexes as an example.

In existing compaction approaches, after copying a valid object from an old segment to a new segment, the cleaner needs to traverse the index, locate and update the reference. When locating an entry, the cleaner starts from the root node and then traverses multiple internal nodes to the deepest leaf node. The pointer-chasing path contains several random accesses, and has much higher latency on PM. According to our analysis (§2.2), these reference update operations constitute the most considerable part of overhead in compaction.

However, we observe that the root-to-leaf path was already traversed when the object was created or updated. To prevent the cleaner from traversing the high latency path again, PACMAN leverages the traversal that was done before.

When creating or updating an object, in addition to insert or update of the reference in the index, service threads also record the address of leaf node which contains the reference in the log. We name this additional information `shortcut`. Objects thereupon have a `shortcut` to their reference in the index. During compaction, cleaners could take the shortcut to quickly locate the leaf node in the index. Then, cleaners still need to search the exact entry in the leaf node. To accelerate the last mile, PACMAN also records the position number of the entry in the shortcut. In this way, the reference search operation is offloaded to service threads but without extra effort. Figure 3 shows an example of a shortcut. The `Node Addr` points to the node, and the `KV Pos` records the entry position number of the array.

Handling shortcut invalidation. In addition, PACMAN needs to handle the possible invalidation of the shortcut in two situations. ① The address of an index entry may change (e.g.,

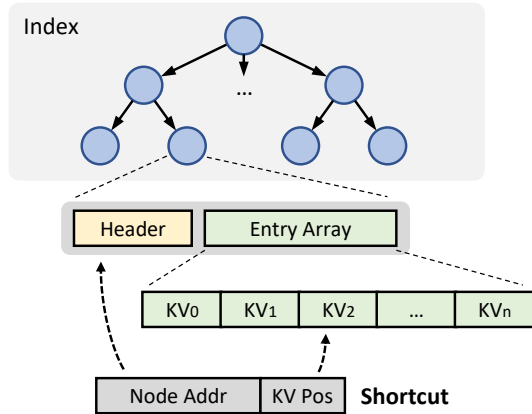


Figure 3: Structure of shortcut (taking a tree-based index as an example). The value in an index’s KV pair is a reference which points to the object in the log. KV Pos is the position number of the associated KV pair (i.e., 2 in this figure).

caused by shift operations in a sorted tree-based index) and thus the shortcut may not point to the original reference. ② The original node pointed by shortcut may have been deallocated, and the original address space may be reclaimed or re-allocated for other usages. Accessing the address wrongly could result in program crash.

The first situation (i.e., Node Addr points to a valid node, but KV Pos is wrong) can be handled easily by conducting some checks. The cleaner will check the header of the node and compare the key indicated by KV Pos to infer that whether the shortcut is correct. These checks are identical as in a normal insert operation. PACMAN attempts to reduce penalty of an incorrect shortcut. For example, in a tree-based index, the cleaner will check if the key still exists in the original leaf node or the sibling node. Thus, shortcuts can tolerate shift operations to a certain extent. If the shortcut is completely invalid, the cleaner updates the reference by falling back to the normal update operation. The penalty of an invalid shortcut is about one or several useless memory accesses and can be further reduced by prefetch technique (§3.3).

To avoid the second situation (i.e., Node Addr does not point to a valid node), instead of directly freeing the space of deleted nodes, PACMAN reserves the deallocated space for future allocations. The deleted nodes are marked as deleted by some means (e.g., a deleted bit in node header or all bytes of the header are set to 0). When creating a new node, PACMAN first attempts to re-use a reserved and same-typed node space. Generally, an index typically has only one or a few fixed types of nodes (e.g., 4 types of nodes in ART [29]), so it is easy to realize the re-allocation. In this way, PACMAN guarantees that the address space of deleted nodes is still valid and this situation turns into the first situation.

Optimizing space overheads. PACMAN stores shortcuts inside the same log segment with their associated objects, which

squeezes the available space and increases compaction pressure. Thus, the benefits of shortcuts are overshadowed especially when the capacity utilization is extremely high and the average object size is small.

PACMAN reduces the space overheads of shortcuts to minimize the punishment of storing shortcuts. First, the size of a shortcut is compressed to 48 bits, including 43-bit Node Addr and 5-bit KV Pos. The Node Addr is compressed based on two opportunities: 1) Current virtual address only uses 47 bit (for user-space virtual address, the 47-63th bits are 0); 2) Memory allocators (e.g., malloc [1], PMDK [4]) allocate objects with at least 16-byte alignment by default. The size of Node Addr can be further reduced according to the specific alignment of nodes in the index (e.g., 512-byte leaf nodes in FastFair). Second, PACMAN doesn’t store shortcuts for objects reclaimed in compaction and hot objects with the help of hot-cold data separation (§3.4). The objects reclaimed in compaction are almost coldest, and the reserved segment are less likely to be compacted again. Hot objects tend to be updated soon and become stale, and shortcuts are useless for stale objects. Besides, shortcut should be disabled when its acceleration cannot cover the punishment of sacrificing more space (e.g., for a hash table-based index and at an extremely high capacity utilization).

Limitation. Shortcut is unsuitable for KV stores with a LSM-based index (e.g., ChameleonDB [50]). Since KV entries in the LSM-based index are moved frequently due to LSM compactions, shortcuts can only stay valid for a transient time.

3.2 Reducing Excessive PM Accesses

Existing garbage collection approaches do not fit persistent memory management as they take no notice of random memory accesses. The introduced small random accesses will cause I/O amplification in PM. To address this issue, PACMAN 1) embeds size information in the reference to reduce PM read, and 2) stores frequently-accessed metadata in DRAM to reduce small random writes on PM.

Embedding size information in the reference. When updating an object, the service thread needs to update the size of total garbage data of each segment for compaction candidate selection. However, getting the size of the stale object needs to read its metadata from PM. To avoid these random PM reads, PACMAN embeds the size of an object in the upper 16 bits of its reference. Therefore, the address and size of the object’s stale version can be acquired from the index together. Because 16 bits can express 64 KiB at most (or larger if objects are allocated obeying to some alignments), PACMAN sets the upper bits of reference to 0 for objects larger than 64 KiB and has to read their metadata when updated or deleted. Fortunately, most objects have small size according to the recent real-world workloads analysis [9].

DRAM-resident garbage information. Since checking references to distinguish the liveness of objects has significant

overhead, especially when the index is on PM, PACMAN turns to deleted flags to store the liveness information. Marking deleted flags in objects' headers on PM will introduce numerous small random writes. To eschew the detrimental effects of small random writes on PM, PACMAN adopts a bitmap on DRAM for each segment. PACMAN locates the corresponding bit of a variable-sized object with its reference directly. We denote the minimum size of an object as `MIN_SIZE` (8 bytes of key, 8 bytes of value, plus the size of metadata header). PACMAN reserves one bit per `MIN_SIZE`. The position of the deleted flag is calculated by dividing the offset within the segment by `MIN_SIZE`. Though this approach leaves some bits unused thereby wasting a small amount of DRAM space, PACMAN can quickly locate the corresponding deleted flags for variable-sized log items. Even in the most extreme case, where the size of the key, value, and header are both 8 bytes, the bitmaps consume DRAM about 0.5% of the log space.

Besides bitmaps, PACMAN stores the size of garbage data of each segment in DRAM which is frequently updated.

3.3 Redesigning the Compaction Pipeline

Traditional compaction algorithms (e.g., memory compaction in RAMCloud and Viper) update the reference right after copying a valid object. Nevertheless, this pattern has several shortcomings on PM without consideration of PM's idiosyncrasies. PACMAN reorganizes the pipeline in a batch pattern as shown in Figure 4. The new algorithm separates the phases of copying objects and updating references, and processes objects in a batched pattern. The cleaner first collects all valid objects from the old segment to a volatile buffer (step ①), and then copies them together to PM (step ②). After that, the cleaner updates their references. Different from using a normal index update operation, the cleaner updates references by a special index update operation (`update_pacman`, lines 43-52) that takes the object's shortcut to locate the reference. Besides, to handle race condition on the index, `update_pacman` carries an extra old value of the reference (`old_addr`) to update the reference in a compare-and-swap semantics (explained later). Subsequently, PACMAN applies the following three optimizations to cater to PM's idiosyncrasies.

(1) Reducing ordering and launch concurrent flushes. In traditional algorithms, for each relocated object, an ordering point is required to ensure that the object has been flushed to PM before updating its reference. These fences are expensive as they stall CPU pipelines.

However, after separating the copying phase and the updating phase, only one fence instruction is needed between the two phases (line 15). PACMAN eliminates the ordering points (`fence`) after update references (step ④). Doing so will not break crash consistency. This is because the old segment is still available until the compaction is finished. Even if an inopportune crash happens before reference updates being flushed to PM (during step ⑤), after restart, for those objects whose references have not been updated or persisted, they can

```

1  NUM_BATCH_FLUSH = 32; // number of concurrent flushes
2  void compact_pacman(Segment segment) {
3      Buffer buffer; // temporal buffer in DRAM
4      vector<ObjectMeta> meta_vec;
5      // iterate objects in this segment
6      for (valid object old_obj : segment) {
7          // ①. generate temporal new object into buffer
8          tmp_new_obj = make_object(old_obj, buffer.offset);
9          buffer.append(tmp_new_obj);
10         meta_vec.push_back(ObjectMeta(old_obj, tmp_new_obj));
11     }
12
13     // ②. copy buffer to reserved segment by ntstore
14     ntstore(reserved_segment, buffer);
15     fence();
16
17     vector<EntryAddr> entry_addr_vec; // for batch persist
18     // iterate valid objects in buffer
19     for (size_t i = 0; i < meta_vec.size(); i++) {
20         // ③. prefetch next object's entry in index
21         prefetch(meta_vec[i + 1].shortcut);
22
23         // ④. update reference
24         (Shortcut, key, old_addr, new_addr) = meta_vec[i];
25         EntryAddr entry_addr;
26         index.update_pacman(Shortcut, key, new_addr, old_addr,
27                             &entry_addr);
28
29         // ⑤. batch persist
30         entry_addr_vec.push_back(entry_addr);
31         if (entry_addr_vec.size() >= NUM_BATCH_FLUSH) {
32             // launch concurrent flushes
33             for (entry_addr : entry_addr_vec) {
34                 persist(entry_addr);
35             }
36             fence();
37             entry_addr_vec.clear();
38         }
39     }
40 }
41
42 // customized index update operation
43 bool Index::update_pacman(Shortcut shortcut, KeyType key,
44                           ValueType new_addr, ValueType old_addr,
45                           EntryAddr *entry_addr) {
46     // find entry by key with the help of shortcut
47     ...
48     *entry_addr = &entry; // record entry address
49     // update only if old_addr matched, e.g., using CAS
50     bool success = CAS(&entry.value, old_addr, new_addr);
51     return success;
52 }

```

Figure 4: Pseudo-code of the PACMAN compaction algorithm and customized index update operation.

still be acquired from the old segment.

Furthermore, PACMAN adopts lazy and batched flushes on reference updates to take advantage of concurrent asynchronous flushes [17], such as `clwb` and `clflushopt`. PACMAN records addresses of updated entries in the index, and launches multiple asynchronous flushes on them, which reduces the average flush latency (step ⑤).

(2) Using non-temporal store to copy valid objects. In copying phase, PACMAN first collects valid objects in volatile segments (step ①), then uses non-temporal store (`ntstore`) to copy them to PM (step ②) for three benefits. First, `ntstore`

has higher bandwidth for large (over 256 B) write than normal store [46]. Second, `ntstore` bypasses the cache and avoids unnecessary cache pollution. Third, `ntstore` can also avoid repeated flushes on the same cache line due to non-aligned writes, which incurs dramatical delay [12].

Though flushes are not necessary for data persistence in new generation CPUs with support of eADR [2], sequential writes (e.g., copying objects) without flushes will turn into random writes on PM due to the random eviction of CPU cache [23]. Therefore, the batch pattern is still beneficial on new CPUs with eADR for adopting `ntstore`.

(3) Leveraging prefetching on PM. Since most valid objects being collected are cold, their references in the index are less likely to stay in cache, which results in high memory access latency, especially for PM indexes. Fortunately, with the shortcut, PACMAN can easily prefetch the index node or entry of the next valid object when updating the current object's reference (step ③), hiding the high access latency with update operation. Thanks to shortcuts and the redesigned pipeline, the reference update operations are quite lightweight, which makes prefetching's effect much more evident.

Handling race condition on the index. The contention between cleaner threads and service threads should be handled properly. A service thread may update an object while a cleaner thread copies the old version of this object and updates its reference. In such a case, the new version of the object updated by the service thread will be covered by the old version of the object. The batched compaction pipeline increases the possibility of this race condition.

One naive approach is holding a lock and blocking service threads during the whole time of relocating an object (lookup reference, copy object, and update reference) [39]. PACMAN minimizes the critical section on updating references, which updates references in a compare-and-swap pattern. Specifically, cleaner threads update references via a customized index update operation (i.e., `update_pacman`) that carries an extra old value of the reference (`old_addr` in step ④). In `update_pacman`, only when the original value of the index entry matches the old value `old_addr`, which means no race condition has happened, the reference can be updated to `new_addr`. If the update fails, the reclaimed object in new segment is marked as stale.

Comparing with FlatStore's compaction pipeline. Though FlatStore [12] also optimize the compaction pipeline by separating the copying phase and the reference updating phase to reduce the fence instructions, it suffer from severe performance issue. Specifically, as FlatStore conducts the index traversal twice for each valid object (i.e., check the liveness and update the reference), the batch pattern exacerbates the overhead on the index. Results in a long reuse distance between the two index traversals for each alive object, causing updating reference not to take the benefit of cache brought by checking reference. By contrast, PACMAN eliminates the first in-

dex traversal by checking in-DRAM deleted flags and reduces the second traversal overhead by prefetching and shortcuts.

3.4 Separating Hot-Cold Data

PACMAN leverages hot-cold data separation to reduce the amount of valid data copying on PM and corresponding reference updates during compaction. Specifically, service threads append hot objects in their per-thread segment and cold objects in another per-thread cold segment. Though hot-cold data separation is a well-known technique to improve garbage collection efficiency [10, 18, 36], we elaborately design how to 1) identify hotspots and 2) handle hotspots shift in the context of the low-latency key-value store on PM.

Identifying hotspots in a lightweight pattern. PACMAN uses a small read-only hash set of hot objects for service threads to distinguish hotspots. The identification of hotspots should be lightweight enough to not increase too much extra latency. Previous work could distinguish the hotspots without maintaining the hotness of objects, such as by data type [28] or hash-based partition [10]. However, these methods are not feasible for PACMAN due to the lack of type semantics or the per-thread logs. Dynamically maintaining hotspots set is not ideal, because it introduces extra operations to another index, resulting in contention and crash thrashing.

Handling hotspots shift and generating new hotspots set. As the access patterns keep changing in real-world workloads, PACMAN uses a lightweight mechanism to detect hotspots and generate new hotspots set. First, service threads keep counting the hit ratio of the old hotspots set to detect whether a hotspots shift has happened. Second, if the hit ratio is lower than a customized threshold which means the hotspots have shifted and the original hot set is stale, service threads record the keys of updated objects in their local circular buffers by sampling. A background thread collects the keys recorded by service threads, sorts these keys using a heap, and generates a new hot set. Third, the background thread changes the pointer of hot set to the new one using CAS. After waiting for a grace period through an RCU-like barrier, the background thread confirms that no service threads are accessing or will access the old hot set, then frees the old hot set safely. Note that the background thread consumes negligible CPU resources as the hotspots shift in real-world workloads is not frequently (at least second-level) [9, 21]. It doesn't matter if service threads identify hot keys inaccurately since it only determines the location of objects. Moreover, cold objects get another chance to be separated from hot objects by compaction.

Though checking the read-only hash set is lightweight, it would bring no benefit but extra overheads in uniform workloads. In the second step above, if the occurrence of the hottest key is close to the occurrence of the coldest key (e.g., less than 3×), the background thread clears the new hot set.

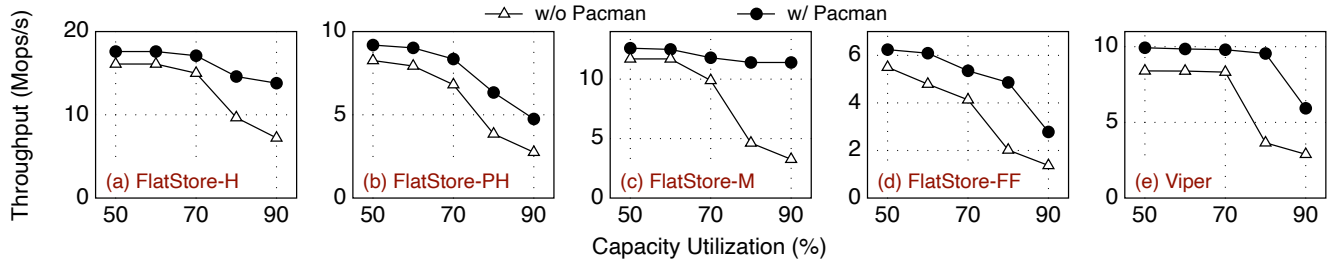


Figure 5: Impact of capacity utilization.

3.5 Recovery

The recovery of the log and the index is similar to existing work [12]. We mainly discuss recovery related to PACMAN.

Shortcut. For KV stores with a volatile index, all shortcuts are invalid after a restart. However, since all segments have to be scanned to rebuild the index, new valid shortcuts are rebuilt in the meantime. For KV stores with a persistent index, to make shortcuts still valid after a restart, PACMAN stores the offset from the base address of the PM pool for the index (e.g., `PMEMobjpool` in PMDK [4]) in the `Node Addr` instead of the virtual address of the node.

DRAM-resident information. Though the DRAM-resident information is unavailable after recovery, the loss of this information has no impact on service threads.

For KV stores with a volatile index, since the index needs to be recovered by scanning all segments, the bitmaps are recovered together with the index at the same time. To distinguish the latest and stale objects, PACMAN compares their version number and retains the latest reference in the index. For KV stores with a persistent index, the volatile bitmaps and metadata only cripple the garbage collection after recovery for a while. Cleaner threads scan the segments and recover the bitmaps by checking references in the index. Then the scanned segments become candidates for compaction.

Crash of compaction. After a restart, unfinished compaction does not need to resume. If a crash happens before the reference updating phase (before line 16 in Figure 4), after a restart, the new segment is still marked as a free segment as nothing has happened. If a crash happens in the middle of the reference updating phase (after line 16 in Figure 4), after a restart, background threads will scan these segments and distinguish redundant objects as described above.

4 Evaluation

In this section, we use a series of experiments to evaluate PACMAN. After describing our setup (§4.1), we first conduct experiments to show the overall performance of PACMAN on PM-based log-structured KV stores under various workloads (§4.2). Then, we analyze the benefit of each optimization of PACMAN with different cases (§4.3). Last, we compare log-structured KV stores with PACMAN against other KV stores on PM with a production workload (§4.4).

4.1 Experimental Setup

All experiments are conducted on a server with Intel Xeon Gold 6240 CPUs. Each CPU has 18 physical cores (36 logical cores with hyper-threading). Each socket is equipped with three 128 GB Intel Optane DC Persistent Memory (DCPMM) DIMMs and 96 GB DRAM. We bind all threads to a single socket to avoid NUMA effect [25, 46]. The Optane DIMMs are configured in App Direct mode.

We apply PACMAN to four versions of FlatStore [12] and Viper [8] we have evaluated in §2.2. FlatStore adopts a log batching technique to reduce the persisting overhead. Viper leverages PM-specific access patterns and employs CCEH [37] in DRAM as its index.

We set the upper limit of the hotspot set size (§3.4) to 128K in all experiments. We co-locate the background thread for generating new hotspot set (§3.4) with a random cleaner thread. The `MIN_SIZE` (§3.2) is set to 32, which means that the deleted flag bitmaps use 1 bit in DRAM for every 32 bytes in segments, which is 0.4% of the whole log size. We use 8-byte keys in all evaluated systems. Unless otherwise stated, we restrict the capacity utilization to 80%. Also, the value size is fixed at 48 for simplicity, because performance mainly depends on the average object size but not the exact size distribution [39] and the value size is corresponding to the recent real-world workloads [9].

4.2 Overall Performance

4.2.1 Impact of Capacity Utilization

We show how PACMAN mitigates the performance decline at high capacity utilization with the same experiments as in §2.2, in which 12 service threads perform write-intensive (50% Put, Zipfian distribution with parameter 0.99) workloads and 4 cleaners conduct the compaction work. Figure 5 shows the results, from which we make three observations.

First, PACMAN obviously curtails the performance decline at high capacity utilization. FlatStore-M with PACMAN maintains throughput above 90% even at extremely high capacity utilization (90%). Due to the huge overhead of compaction, original systems can not fully utilize their performance at high utilization. PACMAN improves the efficiency of compaction according to PM’s peculiarities, and therefore reduces the compaction overhead. Though the performance declines of FlatStore-PH

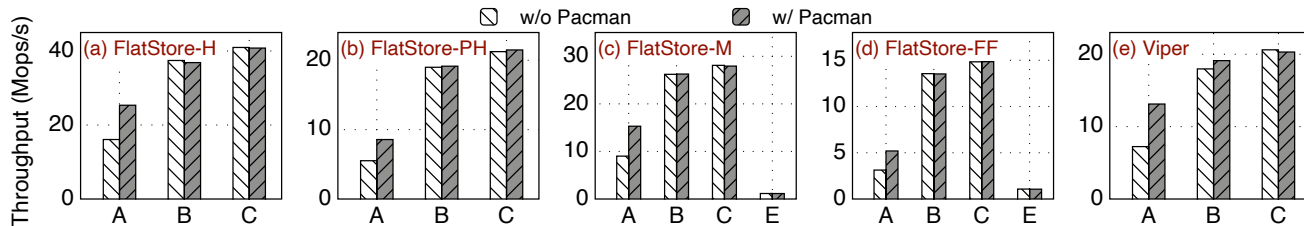


Figure 6: YCSB workloads performance. (*FlatStore-H*, *FlatStore-PH*, and *Viper* don't support scan operations in YCSB-E.)

and FlatStore-FF with PACMAN at 80% utilization still exceed 20%, they are much better than the original systems.

Second, PACMAN also enhances the performance at low capacity utilization, which is because PACMAN reduces small random accesses on PM and saves PM's limited bandwidth. The improvement on Viper is more evident since Viper marks deleted flags and modifies locks on PM.

Third, systems using PACMAN with 4 cleaners also outperform the original systems with 8 cleaner threads (see §2.2). PACMAN saves CPU resources for compaction but brings more improvements, which is not only due to the efficient compaction, but also because of the reduction in contentions on both the index and PM resources [46].

4.2.2 YCSB Benchmark

In this section, we evaluate the basic performance with YCSB [13] benchmark. Table 1 shows the characteristics of workloads. We omit the YCSB-D as it has similar traits to YCSB-B. We set 24 threads to perform workloads after random prefilling 200 million objects, and 4 cleaners for all evaluated systems. Each service thread performs 20 million operations. The capacity utilization is restricted to 80%.

Workload	Feature	Read-Write-Scan %
A	write-intensive	50-50-0
B	read-intensive	95-5-0
C	read-only	100-0-0
E	scan-intensive	0-5-95

Table 1: YCSB workloads description.

The experimental results are presented in Figure 6, from which we have two observations.

First, under write-intensive workload (YCSB-A), PACMAN improves the performance of each system by 1.5-1.8×. PACMAN improves Viper most among these systems, which is because PACMAN on Viper not only increases the compaction efficiency but also reduce small random writes on PM.

In this workload, 4 cleaner threads are insufficient for systems without PACMAN. The cleaner threads' CPU utilizations are all above 95%. For evaluated systems with PACMAN, there are a few unused CPU cycles. For example, with PACMAN the cleaner threads' CPU utilization is 81% in FlatStore-H and 92% in FlatStore-FF. The random prefilling phase invalidates a part of shortcuts. For example, the invalidation ratio of short-

cuts is about 25% in FlatStore-FF and 58% in FlatStore-M. FlatStore-FF has lower invalidation ratio because FastFair has larger leaf node than Masstree and can tolerate more shift operations. Note that we regard the shortcut as valid if the entry can be found in the node indicated by the shortcut or its sibling node.

Second, under read-dominated workloads (YCSB-B, C, and E), systems with PACMAN have similar performance with the original systems. This is because PACMAN does not directly influence read and scan operations, their performance under read-dominated workloads is similar.

4.2.3 Sensitivity Analysis

In this section, we evaluate how workload characteristics affect PACMAN. The default configurations are the same as in §4.2.1. We only show results of FlatStore-H with PACMAN in Figure 7 as a representative sample.

Uniform workloads. In uniform workloads, the system's performance drops compared with that in skewed workloads (Figure 5(a)), which is because of poor locality and much severe garbage collection overhead. The raw FlatStore-H drops more than half at 80% capacity utilization. However, PACMAN still curtails the throughput decline within 15% at 80% utilization and 50% at 90% utilization.

Thread scalability. Due to the compaction overhead, the raw FlatStore-H can not scale well with multiple threads. On the contrary, with only 4 cleaners, FlatStore-H with PACMAN can scale linearly up to about 20 threads.

Value size. The throughput of both systems drops with the value size getting larger because larger objects consume more bandwidth. PACMAN enhances the throughput of FlatStore-H by 50-70% as the value size varies from 32 to 512, which shows that value size has little impact on PACMAN.

Write ratio. Since the compaction overhead increases with write ratio, the improvement of PACMAN on FlatStore-H is more significant with higher write ratio, reaching to 2.3× when the write ratio is 100%. A strange phenomenon is that FlatStore-H with PACMAN has higher throughput when the write ratio is 80% than 60%. We find that cold segments have shorter ages due to the faster write rate. Thus, cleaners are more likely to select hot segments to compact according to the cost-benefit strategy, which has lesser overhead.

Number of objects. With the capacity utilization fixed at 80%, we vary the number of objects to prefill. The number of ob-

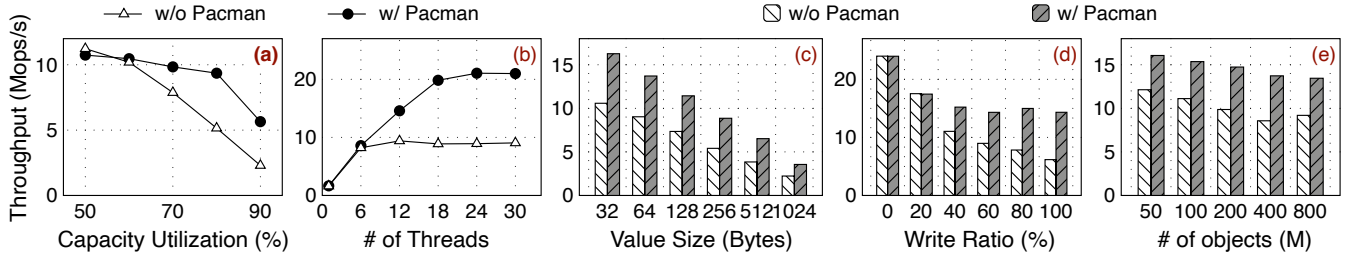


Figure 7: Sensitivity Analysis (on FlatStore-H). (a) Uniform workloads. (b) Thread scalability. (c) Different value sizes. (d) Different write ratios. (e) Different numbers of objects.

jects has two effects on the performance. On one hand, the CPU cache miss rate gets higher with the larger memory footprint (both the index and the log space). On the other hand, as the log space gets larger, the system writes more segments when compaction is triggered and has more time to accumulate stale objects. Thus, the candidate segments for compaction have less live data and the compaction is less expensive. For different numbers of objects, PACMAN can improve the performance by 35-60%.

4.3 Analysis of Techniques

In this section, we analyze the performance benefit of each technique by applying them one by one. To differentiate results more obviously, we stress the system with an artificial workload at a high capacity utilization level (80%) modeled after prior work [39], in which the system throughput is heavily limited by compaction. After loading 200 million KV objects, 12 service threads overwrite these objects following a Zipfian access distribution with skewness parameter 0.99, and 2 cleaner threads perform the compaction work. The workload proceeds for a while until the system throughput and compaction overhead converge to a stable value.

We only show results of PACMAN on FlatStore-H and FlatStore-FF since their results are representative.

Figure 8 shows the contribution of each technique to the throughput and corresponding compaction bandwidth (bytes of cleaned segments per second). The applying order is determined by the dependencies between these techniques (e.g., the shortcut relies on hot-cold separation to reduce its space overhead). Specifically, we gradually apply to the *raw* systems with *reducing* avoidable PM accesses (§3.2), hot-cold data *separation* (§3.4), *shortcut* (§3.1), and the redesigned *batching* compaction pipeline (§3.3).

The results show that all techniques contribute to the improved performance more or less in some cases.

① **Reducing.** Storing garbage information in DRAM and embedding size information in reference eschew the avoidable PM random accesses. Especially in FlatStore, this technique avoids checking references in the index for identifying liveness of objects. Therefore, it brings about 80% improvements on FlatStore with a tree-based index (i.e., FlatStore-FF and

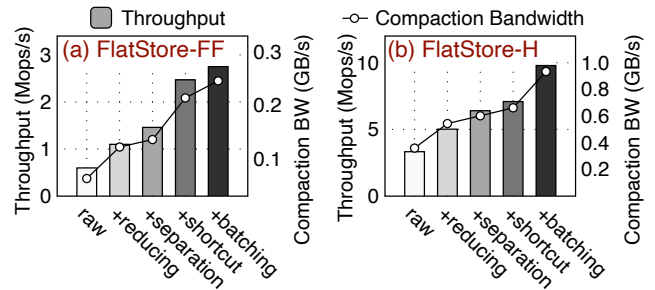


Figure 8: Contributions of techniques to throughput and compaction bandwidth.

FlatStore-M), and about 50% on FlatStore-H.

② **Separation.** Hot-cold data separation improves the system performance about 30%. The improvement of system throughput is more than the compaction bandwidth. This is because that hot-cold data separation alleviates the mixture of stale and valid objects in a segment, thus decreasing the amount of valid objects moving and compaction time.

③ **Shortcut.** Though PACMAN trades some available log space for storing shortcuts, shortcuts still boost the system performance by about 70% for FlatStore-FF and about 45% for FlatStore-M (not shown in the figure) even at high capacity utilization. Moreover, since we do not store shortcuts for hot objects, the rate of successfully using shortcuts to update references is 55% in FlatStore-FF in this experiment. The effect is not obvious for systems with a hash table-based index, since the benefit brought by shortcuts is overshadowed by the additional overhead of storing shortcuts. Since PACMAN only stores shortcuts for inserted cold objects, and we assume that about half of inserted objects are cold, the space overhead is less than 4% for 64B objects. Note that this space overhead has been paid by PACMAN.

④ **Batching.** The batching compaction pipeline and corresponding optimizations bring another 40% improvement on FlatStore-M and FlatStore-H. However, batching has smaller effect on FlatStore with a persistent index, which is because the main overhead of compaction comes from operations on the persistent index.

Put them together, PACMAN increases the compaction bandwidth, and improves the system throughput by about 3× for

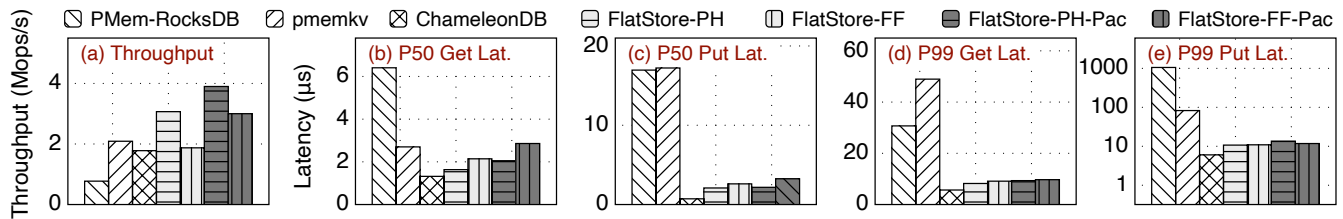


Figure 9: Facebook ETC throughput and latency. (Y axis in P99 Put Lat. is in log scale.)

FlatStore-H and 4.6× for FlatStore-FF.

The results of PACMAN on Viper are similar to FlatStore-H, except for *reducing* and *batching*. As Viper uses deleted flags in objects, *reducing* on Viper has smaller effect than FlatStore. However, due to the inefficiency of Viper’s original compaction, the batching compaction algorithm significantly improves Viper’s performance.

4.4 Comparison with Other KV Stores

In this section, we compare FlatStore with PACMAN (denoted by suffix *-Pac* in Figure 9) against other three representative KV stores on PM, *ChameleonDB* [50], *pmemkv* [6], and *PMem-RocksDB* [3, 5]. *ChameleonDB* adopts a LSM-based hash index tracking KV objects in the log. We implement *ChameleonDB* since it is not open-source. Our implementation can achieve approximate performance in its paper when not considering garbage collection. As *ChameleonDB* does not provide their garbage collection approach, we implement its garbage collection like *WiscKey* [33] for their similar structures, but with a hot-cold data separation (§3.4). *pmemkv* internally leverages *PMDK* [4] for object allocation, which is a non-copying allocator. We set *pmemkv*’s storage engine to *cmap*, a persistent concurrent hash map. *PMem-RocksDB* is based on *RocksDB* [16], a LSM-tree-based key-value store. *PMem-RocksDB* locates SSTables and write ahead log (WAL) on PM. We follow the recommended configurations [5] except for enabling the key-value separation of *PMem-RocksDB*, since it offloads the object management to *PMDK*, which is similar to *pmemkv*. Note that we do not restrict the capacity of *pmemkv* and *PMem-RocksDB*.

We compare these systems with a production workload from Facebook ETC memcached pool [7]. Specifically, the workload has trimodal object size distribution, where the size of an object can be small (1-13 bytes), median (14-300 bytes) and large (larger than 300 bytes). This distribution is representative in real-world productions, as it also resembles the workloads of UP2X at Facebook [9]. We use skewed distribution (Zipfian parameter 0.99) for small and median objects, and uniform distribution for large objects as prior work [12, 14]. After random prefilling each system with 200 million objects, each thread performs 20 million operations of write-intensive workload (50% Get and 50% Put).

For a fair comparison, 1) we only include FlatStore-PH and FlatStore-FF as all compared systems have a persistent

index; 2) we disable the log batching in FlatStore as a normal embedded KV store. All systems use 24 service threads and 4 background threads (for compaction and *PMem-RocksDB*’s flush) except for *pmemkv*. As *pmemkv* does not need garbage collection, we set *pmemkv* with 28 service threads. We report their throughput and latency in Figure 9.

Throughput. From the throughput results, we have the following observations.

The log-structured approach has great performance advantages. Even without PACMAN, FlatStore-PH outperforms *pmemkv* as unordered KV stores, and FlatStore-FF outperforms *PMem-RocksDB* as ordered KV stores. However, the garbage collection overhead overshadowed this advantage. Due to the efficient compaction, PACMAN improves the performance of original systems, especially for FlatStore-FF as it has more severe compaction overheads.

PMem-RocksDB has the lowest throughput even though efforts have been endeavoured to optimize *RocksDB* with persistent memory. This is because *RocksDB* was born for SSD, and some design is not suitable for PM. For example, the memtable, WAL, software caching, and file-based management are less effective when the storage device changes from disk or flash to PM.

Though *pmemkv* does not need garbage collection, it has a low performance. Since the value size varies in this workload, *pmemkv* needs to allocate new object by *PMDK*’s *pmemobj* allocator if the size exceeds previous allocated size.

ChameleonDB has much lower performance than expectation. We also evaluate *ChameleonDB* with unlimited log space (i.e., no compaction) that can achieve similar performance to a KV store with a volatile hash table-based index (not shown in the figure). However, the garbage collection of *ChameleonDB* is much difficult due to its LSM-based index. The LSM-based index inserts a new KV pair to memtable directly without looking for the former entry of the same key. Therefore, cleaners are unaware of the garbage information and have to copy bulk of cold data. Its performance decreases sharply at higher capacity utilization due to the inefficient garbage collection. *ChameleonDB* requires a more elaborate garbage collection approach which considers both KV separation [10, 41] and PM’s peculiarities.

Latency. The median Get latency of each system accords with their index’s overhead. *ChameleonDB* has the lowest latency due to its efficient DRAM-PM-hybrid index.

For PMem-RocksDB, the notorious write stalls in LSM-trees [49] result in high Put tail-latencies. The dilatory flush and compaction not only block foreground write operations, but also lead to multiple SSTable levels, which makes Get operations inefficient. Besides, for low-latency devices like PM, bloom filters aimed at reducing storage I/Os can introduce non-negligible latencies in Get operations.

As pmemkv turns to transaction and logging for atomic in-place updates, it has the highest median Put latency and tail latency among all evaluated systems with a high thread count. First, transactions are more likely to be aborted with a high thread count. Second, the low-performance crash consistency mechanism of pmemkv makes index access inefficient.

4.5 Recovery

We evaluate the recovery with FlatStore-H and FlatStore-FF. We randomly prefill 200 million objects of 256B value size with 80% capacity utilization, in which the log space is 63.3 GB. Then we perform 100 million update operations to disorder all segments. We set 8 threads for the recovery.

For FlatStore-H, garbage information and shortcuts are restored with the volatile index in the meantime. It takes 14 seconds to recover all these things in FlatStore-H. For FlatStore-FF, it takes 102 ms to recover the states of segments. Note that the system is ready for service at this moment. Then, the 8 threads recover information for garbage collection (e.g., deleted flag bitmaps) in background, which takes 37 seconds.

5 Related Work

PM-based KV stores. There has been plenty of research on high-performance KV indexes [11, 20, 26, 27, 32, 37, 38, 43, 50, 51] and KV storage systems [8, 12, 22, 24, 30, 42, 49]. In this paper, we focus on log-structured KV stores on PM.

Log-structured memory storage systems. The log-structured design has been widely adopted in storage systems. RAMCloud [39] is a distributed KV store that uses log-structured memory to achieve high memory utilization. It uses memory to serve requests from clients and disk to store backup copies of data. FASTER [20] designs a hybrid log that spans main memory and storage. Nibble [35] is a concurrent log-structured in-memory KV store that uses a scalable multi-head log allocator with a concurrent index. It can scale up to hundreds of cores with ultra-large volumes of memory. MICA [31] and Segcache [47] are in-memory caching systems using the log-structured approach.

The log-structured approach is also embraced by many persistent memory systems. FlatStore [12], RStore [30], and Viper [8] are all DRAM-PM hybrid KV stores that leverage a volatile index on DRAM and log-structured storage on PM. To reduce small writes on PM, FlatStore proposes pipelined horizontal batching to batch small-sized requests from multiple cores, achieving high throughput without sacrificing low latency. Viper assigns threads to different PM

regions to minimize the thread-to-DIMM ratio, and stores data in DIMM-aligned storage segments. NOVA [45] is a scalable persistent memory file system that maintains separate logs for each inode. LSNVMM [19] is a log-structured transactional memory system that takes advantage of copy-on-write to avoid redo/undo logging.

Optimizations on garbage collection. The main overhead of log-structured storage systems comes from garbage collection. RAMCloud designs an elaborate garbage collection approach to enable high memory utilization. RAMCloud decouples the garbage collection on the memory logs and backups. Hence, memory can have higher utilization and backup disks bear less garbage collection work. In-place updates can reduce the pressure of garbage collection [20, 31]. However, the in-place updates could lead to internal memory fragmentation. Furthermore, for persistent memory, in-place updates without expensive logging cannot guarantee crash consistency. Hot-cold separation is beneficial to garbage collection. Log-structured file systems such as F2FS [28] separate data by their types. HashKV [10] partitions KV objects by hashing. They separate hot and cold objects to some extent. Yang et al. [48] propose a PM-aware garbage collector in JVM. They separate the read-mostly phase and the write-only phase to fully utilize PM bandwidth. However, they adopt PM for merely increasing memory capacity and do not provide persistence guarantee.

The peculiarities of persistent memory make the problem of memory compaction more severe. To the best of our knowledge, PACMAN is the first work that optimizes compaction for log-structured key-value store on PM.

6 Conclusion

Garbage collection overhead in log-structured KV stores becomes more severe on PM. We summarize that the culprit is that existing approaches are unawareness of PM's characteristics. In this paper, we analyzed the inefficiencies of existing compaction approaches in log-structured KV stores on PM. According to the analysis, we design, implement, and evaluate PACMAN, an efficient PM-aware compaction approach for log-structured KV store on PM. PACMAN introduces several techniques to streamline garbage collection with the consideration of PM idiosyncrasies. PACMAN significantly boosts the compaction bandwidth and improves the performance of state-of-the-art systems at high capacity utilization. Our implementation of PACMAN is publicly available at <https://github.com/thustorage/pacman>.

Acknowledgements

We sincerely thank our shepherd Changwoo Min and the anonymous reviewers for their valuable feedback. We also thank Junru Li and Zhe Yang for their help on this work. This work is funded by the National Natural Science Foundation of China (Grant No. 62022051, 61832011), and Huawei.

References

- [1] Aligned memory blocks (the gnu c library). https://www.gnu.org/software/libc/manual/html_node/Aligned-Memory-Blocks.html, 2021.
- [2] eADR: New Opportunities for Persistent Memory Applications. <https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html>, 2021.
- [3] How Intel Optimized RocksDB Code for Persistent Memory with PMDK. <https://software.intel.com/content/www/us/en/develop/articles/how-intel-optimized-rocksdb-code-for-persistent-memory-with-pmdk.html>, 2021.
- [4] Persistent Memory Development Kit. <https://pmem.io/pmdk/>, 2021.
- [5] PMem-RocksDB, A version of RocksDB that uses persistent memory. <https://github.com/pmem/pmem-rocksdb>, 2021.
- [6] pmemkv: Key/value datastore for persistent memory. <https://pmem.io/pmemkv/>, 2021.
- [7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.
- [8] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: An efficient hybrid pmem-dram key-value store. *Proceedings of the VLDB Endowment*, 14(9):1544–1556, 2021.
- [9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.
- [10] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. Hashkv: Enabling efficient updates in KV storage via hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 1007–1019, Boston, MA, July 2018. USENIX Association.
- [11] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. uTree: A Persistent B+-Tree with Low Tail Latency. *Proc. VLDB Endow.*, 13(12):2634–2648, July 2020.
- [12] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20*, page 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.
- [13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.
- [14] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 79–94, Boston, MA, February 2019. USENIX Association.
- [15] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49. USENIX Association, February 2021.
- [16] Facebook. Rocksdb. <https://rocksdb.org>.
- [17] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Mod: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20*, page 775–788, New York, NY, USA, 2020. Association for Computing Machinery.
- [18] Jen-Wei Hsieh, Tei-Wei Kuo, and Li-Pin Chang. Efficient identification of hot data for flash memory storage systems. *ACM Trans. Storage*, 2(1):22–40, February 2006.
- [19] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-structured non-volatile main memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 703–717, Santa Clara, CA, July 2017. USENIX Association.
- [20] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, February 2018. USENIX Association.

- [21] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.
- [22] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, Boston, MA, February 2019. USENIX Association.
- [23] Anuj Kalia, David Andersen, and Michael Kaminsky. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20*, page 105–119, New York, NY, USA, 2020. Association for Computing Machinery.
- [24] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for nonvolatile memory with Nov-eLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, July 2018. USENIX Association.
- [25] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for multi-tiered memory systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 715–728. USENIX Association, July 2021.
- [26] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21*, page 424–439, New York, NY, USA, 2021. Association for Computing Machinery.
- [27] R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. TIPS: Making volatile index structures persistent with dram-nvmm tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 773–787. USENIX Association, July 2021.
- [28] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association.
- [29] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49, 2013.
- [30] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. Enabling low tail latency on multicore key-value stores. *Proc. VLDB Endow.*, 13(7):1091–1104, March 2020.
- [31] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association.
- [32] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.*, 13(10):1147–1161, April 2020.
- [33] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, February 2016. USENIX Association.
- [34] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12*, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery.
- [35] Alexander Merritt, Ada Gavrilovska, Yuan Chen, and Dejan Milojicic. Concurrent log-structured memory for many-core key-value stores. *Proc. VLDB Endow.*, 11(4):458–471, December 2017.
- [36] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random write considered harmful in solid state drives. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, San Jose, CA, February 2012. USENIX Association.
- [37] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, Boston, MA, February 2019. USENIX Association.

- [38] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16*, page 371–386, New York, NY, USA, 2016. Association for Computing Machinery.
- [39] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies, FAST'14*, page 1–16, USA, 2014. USENIX Association.
- [40] Denis Serenyi. Cluster-level storage at google. In *Keynote at the 2nd Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Intensive Computing Systems*, 2017.
- [41] Chen Shen, Youyou Lu, Fei Li, Weidong Liu, and Jiwu Shu. Novkv: Efficient garbage collection for key-value separated lsm-stores. 2020.
- [42] Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. TH-DPMS: Design and Implementation of an RDMA-Enabled Distributed Persistent Memory Storage System. *ACM Trans. Storage*, 16(4), oct 2020.
- [43] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A Black-Box approach to NUMA-Aware persistent memory indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 93–111. USENIX Association, July 2021.
- [44] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. Nvm-cached: An nvm-based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys '16*, New York, NY, USA, 2016. Association for Computing Machinery.
- [45] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.
- [46] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.
- [47] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Seg-cache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518. USENIX Association, April 2021.
- [48] Yanfei Yang, Mingyu Wu, Haibo Chen, and Binyu Zang. Bridging the Performance Gap for Copy-Based Garbage Collectors atop Non-Volatile Memory, page 343–358. Association for Computing Machinery, New York, NY, USA, 2021.
- [49] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31. USENIX Association, July 2020.
- [50] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. Chameleondb: A key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems, EuroSys '21*, page 194–209, New York, NY, USA, 2021. Association for Computing Machinery.
- [51] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, Carlsbad, CA, October 2018. USENIX Association.



Towards Latency Awareness for Content Delivery Network Caching

Gang Yan

SUNY-Binghamton University

Jian Li

SUNY-Binghamton University

Abstract

Caches are pervasively used in content delivery networks (CDNs) to serve requests close to users and thus reduce content access latency. However, designing latency-optimal caches are challenging in the presence of *delayed hits*, which occur in high-throughput systems when multiple requests for the same content occur before the content is fetched from the remote server. In this paper, we propose a novel timer-based mechanism that provably optimizes the mean caching latency, providing a theoretical basis for the understanding and design of latency-aware (LA) caching that is fundamental to content delivery in latency-sensitive systems. Our timer-based model is able to derive a simple ranking function which quickly informs us the priority of a content for our goal to minimize latency. Based on that we propose a lightweight latency-aware caching algorithm named LA-Cache. We have implemented a prototype within Apache Traffic Server, a popular CDN server. The latency achieved by our implementations agrees closely with theoretical predictions of our model. Our experimental results using production traces show that LA-Cache consistently reduces latencies by 5%-15% compared to state-of-the-art methods depending on the backend RTTs.

1 Introduction

Content delivery networks (CDNs) carry more than 50% of today's Internet traffic [17] by caching a variety of contents such as videos, music, software downloads, etc. and delivering thousands of millions of user requests each day. CDNs deploy hundreds of thousands of servers across the world to serve user requests. If the requested content is available in the server near the user, a cache hit occurs and the user experiences a quicker response with a lower latency. Otherwise, a cache miss occurs and the requested content has to be fetched from the remote server with a dramatically increased latency. As a result, there has been a renewed focus on increasing cache hits [11, 16, 31, 40], which can significantly improve the content delivery of the Internet.

The recent trends of improving caching efficiency in terms of *maximizing caching hits* mostly focus on designing different content caching algorithms, including but not limited to GDSF [15], ARC [44], CAR [6], LHD [7], A-LRU [37], AdaptSize [11], CACA [29], LRB [53], RL-Cache [35], RL-Bélády [58], DeepCache [45] and LHR [59]. However, most of these algorithms assume that the user-perceived latency upon a cache hit is *negligible* (i.e., zero delay). Though this assumption has been widely used in the caching literature, some recent efforts start linking it to the potential performance degradation when *minimizing the end-user latency* in the presence of *delayed hits* [27, 55]. Notably, the latest series of works [4, 42] reveals that a *delayed hit* can occur in real-world systems, especially in high-throughput systems when multiple requests to the same content occur before the requested content is fetched from the remote server. As a result, the aforementioned caching algorithms fail to minimizing user-perceived latency in the presence of delayed hit since they are designed under the assumption that delayed hits does not exist. See Section 2 for more detail.

Despite the insightful findings in [4, 42], there remains a major *gap* between the delayed hits observation and the goal of efficient *online* latency-aware caching algorithm design. This is due to the fact that delayed hits were *only* identified and overcome through a hard *offline* optimization problem assuming that all contents are the *same size*, and the fetching latency from remote server upon a cache miss is *uniform* across different contents. However, it is well-known that content sizes often vary widely in production CDNs from a few bytes [46] to several gigabytes [31]. Additionally, the fetching latencies upon cache misses in production systems may vary over time due to the network conditions (e.g., bandwidth), and differ across content sizes since large contents often require longer fetching latencies (e.g., multiple RTTs). These facts introduce an additional layer of complexity in the design of *online* algorithms for minimizing latency in the presence of delayed hits, which largely remain elusive in the literature.

This paper closes this gap by developing a novel and lightweight timer-based mechanism to account for the impact

of delayed hits for contents with variable sizes and different fetching latencies that provably optimizes the mean caching latency. This provides a theoretical basis for the understanding and design of latency-aware caching that is fundamental to content delivery in latency-sensitive systems. In this approach, each content is associated with a timer indicating the fetching latency between the cache and remote server, which can be variable across different contents and systems. Upon a cache miss, all requests (i.e., delayed hits) arriving at the cache during a certain time period dictated by its timer suffer a corresponding latency before these requests are truly served. This approach is able to explicitly characterize the expected average latency of a caching system in the presence of delayed hits. This further enables us to derive a simple ranking function which can quickly prioritize contents for the purpose of minimizing latency.

This paper makes the following research contributions:

- To the best of our knowledge, our proposed timer-based model is the first to provide a theoretical basis for understanding the impact of delayed hits in latency-sensitive caching systems in an online manner. This enables us to design a lightweight online latency-aware caching algorithm which can capture the variable fetching latency and different content sizes in real-world systems.
- Using this timer-based model, we explicitly characterize the mean latency of each content in the presence of delayed hits and derive a simple ranking function to prioritize contents so as to minimize the mean latency. We then propose a lightweight latency-aware caching algorithm named LA-Cache.
- We have implemented the LA-Cache prototype within Apache Traffic Server, a popular CDN server, and evaluate the performance of LA-Cache using production traces. Our empirical results are in close alignment with our theoretical model predictions. Furthermore, we show that LA-Cache consistently outperforms conventional caching algorithms by reducing the latency by 5%-15% depending on the backend RTTs.

The rest of the paper is organized as follows. We introduce the motivations and opportunities in designing latency-aware caching in Section 2. We present the model for delayed hits and explicitly characterize the mean latency in Section 3. We propose the latency-aware algorithm LA-Cache in Section 4, and present its prototype design in Section 5. Evaluation results are shown in Section 6. We discuss related work in Section 7 and conclude the paper in Section 8. Additional results are presented in the supplementary material.

2 Background and Motivation

We begin by motivating the existing of delayed hits in latency-sensitive systems and showing the fundamental limitations of

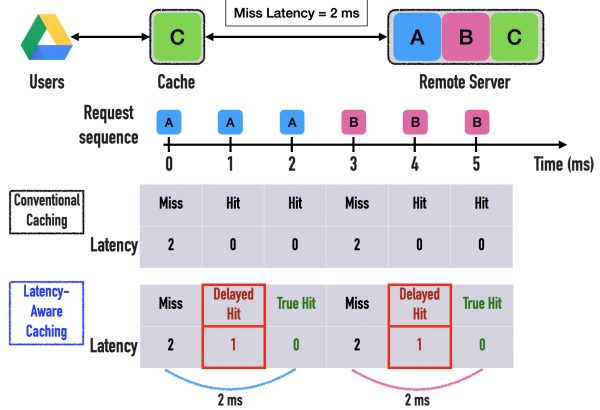


Figure 1: A motivating example for latency-aware caching where delayed hits occur. Suppose that the remote server stores all three contents named A, B, and C, the cache size is 1, and the miss latency (i.e., the Round-trip Time (RTT)) between the cache and the remote server is 2 ms. We also have a request sequence of A,A,A,B,B,B, ... with a new request arriving at the cache every 1 ms.

existing algorithms.

2.1 A Motivating Example

As a motivating example, we consider a basic delayed hit scenario in real-world systems as shown in Figure 1. Upon the first request to content ‘A’, a cache miss occurs and the cache must fetch content ‘A’ from the remote server with a latency of 2 ms. The next two requests to content ‘A’ can be directly served from the cache and experience a latency corresponding to a cache hit, which is assumed to be zero in conventional caching algorithms. Similar process happens for the requests to content ‘B’. Since it takes 2 ms to fetch content ‘A’ from the remote server, how is it possible for the second request to content ‘A’ that arrived just 1 ms after the miss was served with a zero latency? Clearly, something is wrong.

Contrast to the ideal assumption in conventional caching algorithm design, the following *actually happens*. Upon the first request to content ‘A’, a cache miss is claimed and a fetch is triggered with a latency of 2 ms. Since the RTT is 2 ms, content ‘A’ will only arrive in the cache at time $t=2$ ms. As a result, the request at $t=1$ ms is queued behind the original miss, and must wait (at least) 1 ms to be served. At time $t=2$ ms, content ‘A’ arrives at the cache and all queued requests (including the one just arrived) are resolved. Similar process happens for the requests to content ‘B’. These requests (e.g., requests to content ‘A’ at $t=1$ ms and to content ‘B’ at $t=4$ ms) are called *delayed hits* since they neither suffer the latency of a cache miss¹ nor that of a true cache hit (e.g., requests to

¹We interchangeably use the terms of “miss latency”, “fetching latency”

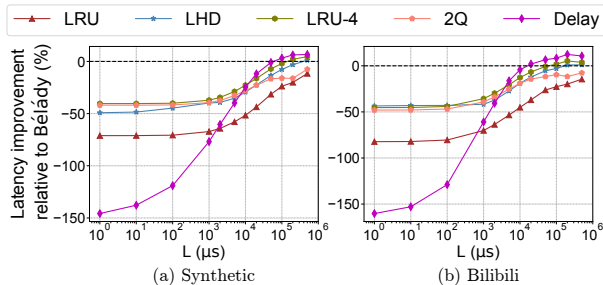


Figure 2: Conventional caching algorithms fail to minimize latency: As the fetching latency (the value of L) increases, conventional caching algorithms outperform the Bélady’s offline MIN algorithm in term of mean latency on a 256GB cache. For example, LRU-4 outperforms Bélady when L is greater than 50 ms in the Bilibili trace.

content ‘A’ at $t=2$ ms and to content ‘B’ at $t=5$ ms).

In general, delayed hits occur in *high-throughput systems* when multiple requests to the same content occur before the requested content is fetched from the remote server [4, 27, 42, 55]. This phenomenon has become increasingly perceptible due to the growing ratio between system throughputs and latencies in a wide range of systems. For example, the wide-area latency between a CDN forward proxy and a central data center is only marginally improving, while newer technologies boast order-of-magnitude throughput improvement with the network links moving from 10 Gbps to 400 Gbps [24]. The fundamental problem is that latencies are closer and closer to limits imposed by the speed of light, while throughputs keep growing unboundedly. Hence, minimizing the impact of delayed hits is a key performance objective in latency-aware caching systems.

2.2 Limitations of Existing Algorithms

The above example indicates that conventional caching algorithms fail to capture the impact of delayed hits, which can be significant especially in systems with high latency to the remote server. One reason contributes to this failure is that conventional caching algorithms are designed to maximize cache hits under the assumption that all cache hits result in zero delay, i.e., they equally treat delayed hits and true hits. To that end, the latency measured by conventional caching algorithms significantly *underestimate true latency* in the presence of delayed hits. Some so-called “cache hits” will experience latencies closer to the high latency of a cache miss than the low latency of a true hit in practice.

As a consequence of this discrepancy, conventional caching algorithms fail to minimize latency although some caches were deployed for this purpose, which were actually treated equivalently to maximize cache hits regardless of delayed hits.

and “RTT” in this paper.

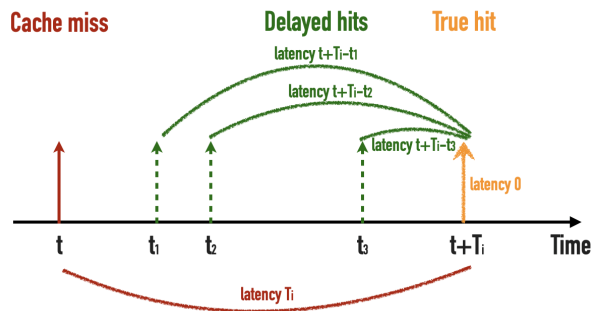


Figure 3: The illustration of our proposed timer-based model for a particular content i in the presence of delayed hits.

For example, the Bélady’s offline MIN algorithm² [8] is the well-known optimal algorithm in maximizing cache hits and minimizing latency when all contents are of the same size and delayed hits are treated as true hits. However, Bélady is no longer optimal in minimizing latency in the presence of delayed hits, as illustrated in Figure 2 (More details in Section 6). Therefore, *the goal of maximizing cache hits for most conventional caching algorithms and the goal of minimizing latency are not equivalent, in the presence of delayed hits*³. We need to design new online algorithms for latency-sensitive caching systems.

3 Model for Latency Minimization

We consider the problem of minimizing latency for content delivery given delayed hits in latency-sensitive systems. In particular, we aim at designing a latency-aware caching policy that minimizes *the mean latency of all requests*. For ease of exposition, we denote the latency to fetch a particular content i from the remote server as L_i , which can vary across contents and servers. Therefore, the latency is (a) L_i upon a cache miss, (b) between 0 and L_i upon a delayed hit, and (c) 0 upon a true cache hit.

3.1 Modeling Delayed Hits

Our key insight is to build a novel connection between delayed hits and the timer-based caching policy. Figure 3 shows such a novel connection for a particular content i . More specifically, each content i has a timer value L_i which represents the latency to retrieve the content from the remote server upon a cache miss. In other words, upon a cache miss at time t , a request is sent to the remote server and content i will be

²The Bélady’s offline MIN algorithm always evicts the content with the furthest next request.

³It has been shown [42] that the latency objective is not antimonotone for caching problems with delayed hits. In other words, a caching algorithm that improves average caching latency under delayed hits might even lower the true hit rate. Hence optimizing caching latency is fundamentally different from optimizing cache hits.

fetched and inserted into the cache after L_i time slots, i.e., at time $t + L_i$. The timer counts down and until the timer expires, any new requests for content i during $[t, t + L_i)$ are called *delayed hits*. These requests are queued behind the original cache miss occurred at time t , and resolved at time $t + L_i$ with a corresponding latency of $t + L_i - t'$ for the request occurred at time t' . The request arrives at time $t + L_i$ is satisfied immediately with a latency of 0, and hence is called the true hit.

3.2 Mean Latency of All Requests

Suppose that the requests for content i arrive at the cache according to a Poisson process⁴ with rate λ_i . Let D_i be the expected aggregated latency experienced by the arrival of requests for content i (i.e., delayed hits) upon a cache miss. The probability of not finding content i in the cache is the cache miss probability m_i , which satisfies $m_i = \frac{1}{1 + \lambda_i L_i}$ derived in the context of timer-based caches [25].

Proposition 1. *The expected latency experienced by the request arrival of content i is $\left(1 + \frac{1}{1 + \lambda_i L_i}\right) \frac{L_i}{2}$.*

Proof. We first compute the expected aggregated latency for content i . Upon a cache miss, the request is sent to the remote server and content i is fetched and inserted into the cache after L_i time slots. Thus, the latency of a cache miss is L_i . A delayed hit occurs for every arrival of content i during time $[t, t + L_i)$ since the content i has not been inserted into the cache yet, and the corresponding latency for request at time $t' \in [t, t + L_i)$ is $t + L_i - t'$. Thus *the expected aggregated latency by all delayed hits* in the interval of $[t, t + L_i)$ is

$$D_i := \int_t^{t+L_i} \lambda_i(t + L_i - x) dx = \frac{\lambda_i L_i^2}{2}. \quad (1)$$

Since the requests for content i follow a Poisson process, the expected number of requests, i.e., the expected number of delayed hits is $\lambda_i L_i$. Hence, the expected latency for each delayed hit is $\frac{\lambda_i L_i^2}{2} / \lambda_i L_i = \frac{L_i}{2}$. Then *the mean latency experienced by the requests of content i* is a weighted sum of the latency from the cache miss and the latency from delayed hits, which satisfies

$$\bar{D}_i = m_i L_i + (1 - m_i) \frac{L_i}{2} = \left(1 + \frac{1}{1 + \lambda_i L_i}\right) \frac{L_i}{2}. \quad (2)$$

□

⁴Poisson arrivals are widely used in the literature, e.g., [32, 36, 38, 41, 43]. However, our model holds for general stationary process [5] at the cost of complicated notations [25, 26]. We relax the Poisson arrivals assumption for our algorithm design, implementation and empirical evaluation.

Corollary 1. *The mean latency experienced by all requests to N distinct contents satisfies*

$$\bar{D} := \sum_{i=1}^N \lambda_i \bar{D}_i = \sum_{i=1}^N \frac{\lambda_i L_i}{2} \left(1 + \frac{1}{1 + \lambda_i L_i}\right). \quad (3)$$

Remark 1. *The main advantage of our proposed timer-based model for delayed hits is that it can be easily deployed in latency-sensitive systems to provide a precise and theoretically-validated latency. Although real-world systems do not have strict Poisson arrivals for any content (see Section 6), we will show in Section 6 that our proposed theoretical model with Poisson assumptions works well in practice.*

Remark 2. *Timer-based caches have been extensively studied in the community [9, 14, 23, 25, 26, 33, 48–50] which are used to store frequently requested contents in computer systems. In a timer-based cache, a timer value is set when a content is first cached and evict the content when the timer expires. While our timer-based delayed hits model serves a different purpose, some of the theoretical analysis of timer caches directly apply (e.g., the decoupling nature of contents in timer cache analysis). This novel connection between timer-based delayed hits and traditional timer-based caches allows us to bring to bear the analytical work done in the conventional caching domain into latency-sensitive systems.*

4 Latency-Aware Cache

Our novel and lightweight timer-based model for delayed hits provides us an opportunity to design a *latency-aware* (LA) caching policy that achieves optimal latency for any given request sequence with variable content fetching latency from remote server.

Having characterized the mean latency experienced by all requests (see Corollary 1), we turn to derive a simple *ranking function* that can quickly prioritize contents so as to minimize latency⁵. Ranking function has been widely used in the design of conventional caching algorithms. For example, the classic Least Recently Used (LRU) [18] (or its variants) which are employed in major CDNs today, is based on a ranking function of content request recency, while Least Frequency Used (LFU) ranks contents by how frequently they have been

⁵We build a novel connection between delayed hits and timer-based caches for the sake of characterizing the mean latency of each content in the presence of delayed hits. See Remark 2. In particular, the fetching latency upon a cache miss is analogous to a timer. Need to mention that we are not considering the conventional timer-based caches, where each content is decoupled by the timer. Instead, our timer-based model serves a different purpose. For our latency-aware caching policy design, all contents are coupled by the cache capacity constraint and hence a ranking function (which is derived using the timer-based model, see equation (4)) is needed to make caching decisions. We use the term “timer-based model” since we can bring some analytical results from traditional timer-based cache domain into latency-minimization analysis, such as the expression of m_i .

requested. However, all these ranking functions prioritize contents for maximizing cache hits whereas we seek a ranking function so as to minimizing latency.

Our ranking function is inspired by the theoretically-sounded latency derived from the timer-based model for delayed hits, which in particular the following two intuitions. First, we consider the metric of aggregated latency computed in (1), which is the sum of latency due to a cache miss and any delay hits in the next L_i time slots which occur before the corresponding content is fetched and inserted into the cache. Intuitively, a content with a higher latency cost increases the average latency more than a content with a lower latency cost, and hence should be prioritized. Second, we consider the metric of mean latency for each request of a content, which is computed in (2). It is clear that a burst of requests to a content can contribute more to the average latency than a sparse of requests to a content. Following these two intuitions, we derive a ranking function based on both the aggregated latency and the mean latency for each content i as

$$\tilde{f}(i) = \frac{\text{Aggregated latency}}{\text{Mean latency for each request}} = \frac{D_i}{\bar{D}_i} = \frac{\lambda_i L_i (1 + \lambda_i L_i)}{2 + \lambda_i L_i}. \quad (4)$$

Now we introduce the *latency-aware* policy based on the above ranking function, abbreviated as LA-Cache. LA-Cache always places in the cache the C contents with the largest value of their corresponding ranking functions, i.e., if $\tilde{f}(1) \geq \tilde{f}(2) \geq \dots \geq \tilde{f}(N)$, then contents $1, 2, \dots, C$ are cached given that the cache size is C .

Proposition 2. *LA-Cache achieves the minimum mean latency experienced by all requests compared to any other online policies.*

Proof. This is clear from the definition of LA-Cache policy since both the ranking function (4) and the expected latency for each content i in (2) (resp. (3)) are monotonically increasing in $\lambda_i L_i$. \square

Remark 3. *Given (3) (resp. (2)), it can be easily shown that \bar{D} (resp. \bar{D}_i) is increasing in $\lambda_i L_i$, which can be interpreted as the expected number of delayed hits upon a cache miss for content i . Since our ranking function (4) is also increasing in $\lambda_i L_i$, it is clear that LA-Cache prioritizes caching bursty contents (We will formally define and evaluate the burstiness of a content using a burstiness measure [34] in Section 6). Intuitively this is correct since a bursty content usually refers to a large number of requests in a shorter time period. This will result in a larger aggregated latency for the content upon a cache miss. As a result, prioritizing such a content can reduce the latency.*

4.1 From Theory to Practice

The above ranking function is defined under the assumption that the content sizes in the system are equal. However, the

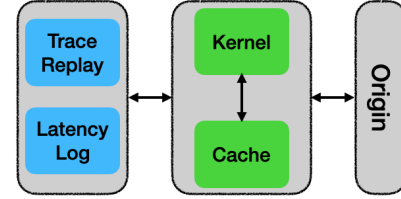


Figure 4: The architecture of our ATS prototype.

content sizes in production system usually vary significantly from a few bytes [46] to several gigabytes [31]. To overcome this drawback, we redefine the ranking function by incorporating the content size s_i , satisfying

$$f(i) = \frac{\tilde{f}(i)}{s_i} = \frac{\lambda_i L_i (1 + \lambda_i L_i)}{(2 + \lambda_i L_i) s_i}. \quad (5)$$

For the notation abuse, we call the policy using this refined ranking function as LA-Cache in the rest of the paper.

To compute the ranking function and obtain the ranking function based policy LA-Cache, the content arrival rate λ_i is needed⁶. This is straightforward for synthetic workload; however, the content arrival rate is usually unknown and varying over time in real-world systems. To this end, we use estimation techniques to approximate the request rates. For any content i , let X_j^i denote the random variable corresponding to the inter-arrival times for the requests for content i , and \bar{X}^i be its mean. We can approximate the mean inter-arrival time as $\hat{X}^i = \sum_{j=1}^K X_j^i / K$. It can be easily shown that \hat{X}^i is an unbiased estimator of $1/\lambda_i$. However, keep tracking of all X_j^i 's for each content j from the very beginning will increase the overhead. As it is well-known that content request processes in production systems are highly dynamic and non-stationary, we further consider a sliding time window, and only use the X_j^i 's within the window to estimate \hat{X}^i . In Section 6, we will use this estimator to compute the ranking function for evaluating our algorithm.

5 Implementation

We implement the LA-Cache prototype within Apache Traffic Server (ATS) [2], a popular CDN server. An LA-Cache cache simulator has also been implemented for the sake of comparison with a wide range of state-of-the-art caching algorithms. The two implementations are written in C++.

5.1 LA-Cache Prototype

ATS is a multi-threaded and event-based CDN caching server with a space-efficient in-memory lookup data structure as

⁶Again, L_i indicates the content retrieval latency upon a cache miss (related to RTTs of the system). In our experiments, we evaluate the impact of its value on the system performance. See Section 6.

	Description
LRU	Recency-based heuristic.
LRU-K	Recency-based heuristic. Evict content with the oldest K-th reference in the past.
LHD [7]	Using a ranking function of the content expected hit density.
2Q [52]	Manage caching decisions through a FIFO queue and a LRU queue.
Delay	An offline heuristic knows future request latency caused by evicting a content from cache now.
LRU-MAD [4]	Calculate content average latency from history and then combine with LHD as a ranking function.
LHD-MAD [4]	Calculate content average latency from history and then combine with LRU as a ranking function.

Table 1: Overview of state-of-the-art caching algorithms.

an index to the cache. A typical ATS configuration consists of a disk/SSD cache and a memory cache. To achieve high performance, ATS is accessed using asynchronous I/Os. The overview of our LA-Cache prototype is presented in Figure 4.

Upon a new request, ATS implements the following steps. Based on the URL, it looks up the local caches to check whether the corresponding content is available. If the requested content is already in the caches (i.e., a true hit), then the request is immediately satisfied by replaying a response back to the user. Otherwise, the request is sent to the kernel, which maintains the request history received from users, i.e., a separate queue for each cache miss (see Section 1). If the current request belongs to one queue, then it will be added to the queue (i.e., a delayed hit). Otherwise, the kernel sets up a new queue for the content (i.e., a miss), and the request is forwarded to the original remote server. To deal with real traces, the requests are sent to a proxy server in the recorded order (via the trace replayer). All users and the master server communicate with each other by the TCP protocol.

We implement LA-Cache on top of ATS. To do so, we replace the lookup data structures for ATS cache with the LA-Cache described in Section 4. The content admission⁷ and look-up processes can be implemented asynchronously. These two processes are used to update parameters so as to make eviction decision⁸. In particular, the eviction process is run by scheduling cache admissions in a lock-free queue. It implements eviction rule to select one eviction candidate when the cache is full. But as for the flash abstraction layer which is very important in production system (i.e., we have

⁷Content admission decides whether to cache the content upon a cache miss.

⁸Eviction process determines which content to evict when the cache is full.

Dataset	CDN-A	CDN-B	Bilibili	Wiki
Duration (Hours)	24	9.9	18.7	0.1
Unique contents	330,446	162,104	4,852	407,919
Total requests (Millions)	0.97	1	1	1
Mean content size (MB)	25.5	68.4	563.5	69.8
Max content size (MB)	7,790	38,392	565.8	3,840

Table 2: Key characteristics of the production traces used throughout our evaluation spanning different CDNs.

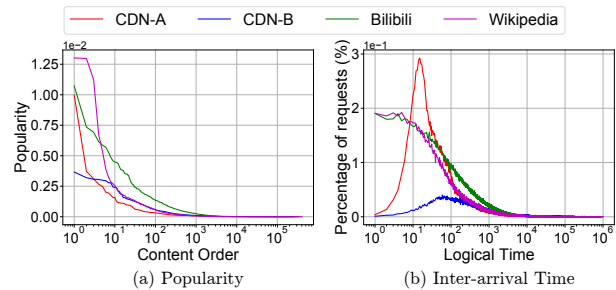


Figure 5: The production traces used in our evaluation comes from different CDNs and thus exhibit different request patterns.

no access, e.g., RIPQ [56]), we only emulate the workings due to some difficulties, reading offsets randomly and writing sequentially to the disk. Since the memory cache is usually small which has little impact on hit probability [11], we keep this part of ATS unchanged. In summary, we implement the framework by only modifying about 100 lines of codes in ATS. The LA-Cache framework library contains about 600 lines of codes.

5.2 LA-Cache Simulator

We implement an LA-Cache simulator that includes a wide range of conventional caching algorithms. For ease of exposition, we only report the results for the “best-performing” algorithms as summarized in Table 1. Finally, our implementation benefits from existing caching simulators such as lib-CacheSim [39] and LRB simulators [53].

Availability. The code for the prototype design, the cache simulator as well as all evaluations in Section 6 are available at <https://github.com/GYan58/la-cache>.

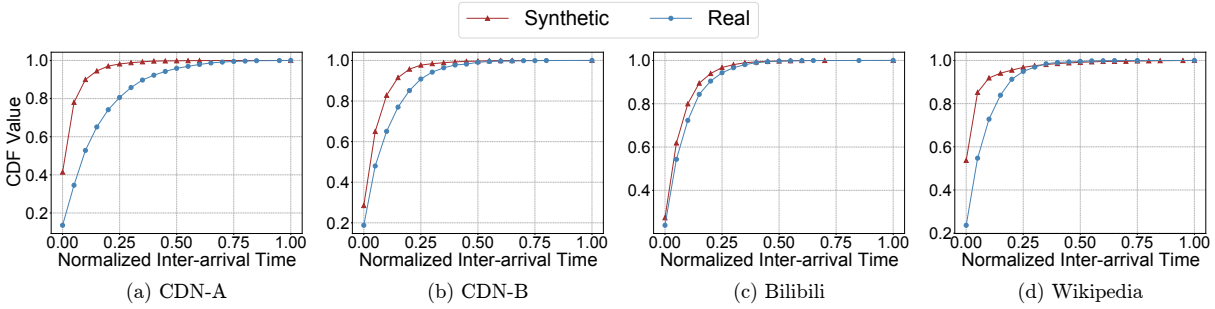


Figure 6: CDF of inter-arrival times for real trace and synthetic trace for two representative contents.

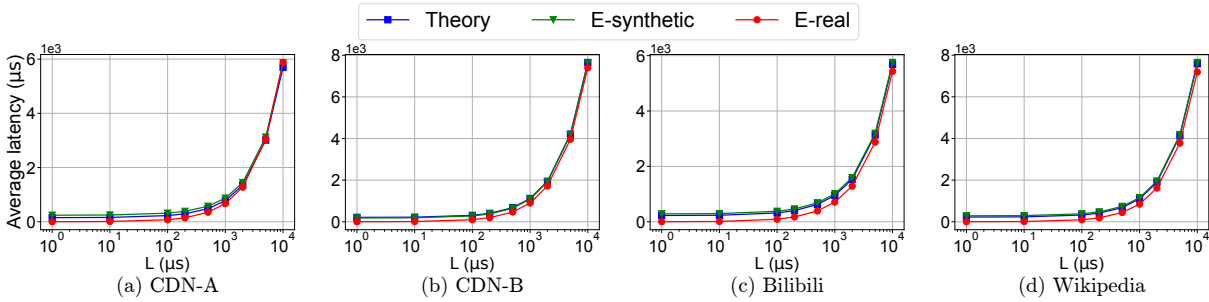


Figure 7: Comparison between theoretical and empirical results of mean latency for production traces. The mean latency graph for the *Theory*, the *E-real* and the *E-synthetic* is slightly offset for ease of visualization.

6 Evaluation

In this section, we evaluate our LA-Cache prototype. We also conduct simulations to compare LA-Cache to a wide range of state-of-the-art algorithms using production traces. Our results address the following questions:

- How accurate is our timer-based model for delayed hits given that real-world systems do not have strict Poisson arrivals (Section 6.2)?
- What is the benefit of using our LA-Cache prototype compared to existing CDN production systems in terms of latency and implementation overhead (Section 6.3)?
- What is the performance of LA-Cache compared to state-of-the-art algorithms on a wide range of production CDN traces under various cache settings (e.g., different fetching latency and cache sizes) (Section 6.4)?

6.1 Methodology

Traces. We consider production traces from four CDNs, two of which chose to remain anonymous. (1) CDN-A collected from several nodes in one continent serves a mixture of web and video traffic; (2) CDN-B captures mobile video behaviors collected from one live streaming system; (3) Bilibili [12, 29], collected from a Video-on-Demand (VoD) provider with millions of HTTP requests; and (4) a Wikipedia (Wiki) trace [53] collected on a west-coast node serving photos and other

media content. We summarize the trace characteristics in Table 2 and present two key distributions of these traces in Figure 5. The traces typically span several tens to hundreds of thousands of requests, and tens of thousands of contents with sizes varying from 10KB to 10^4 MB. The total bytes requested are on the order of TBs; however, the active bytes⁹ are on average on the order of GBs. As a result, we choose the cache size in the range of 128GB to 1,024GB for different traces in our evaluation. For ease of readability, we only present results using a 256GB cache and a 512GB cache for each trace in the rest of this section. Similar observations hold for other cache sizes and hence are omitted. Finally, the average inter-request time is 6.5 ms for CDN-A, 9.1 ms for CDN-B, 1.1 ms for Bilibili and 5.4 ms for Wikipedia.

Baselines. We compare LA-Cache with a wide range of state-of-the-art algorithms. For ease of exposition, we only show the few “best-performing” algorithms (see Table 1) in the following figures.

Performance evaluation. We evaluate the performance of these algorithms using four production workloads described above with different fetching latencies and cache sizes. All results are generated by running on Ubuntu 18.04 with an Intel(R) Core(TM) i7-6700HQ processor and a 8GB RAM.

⁹A content is said to be active at time t in a trace if t lies between its first and last requests. The total size of active contents at time t is defined as active bytes [35].

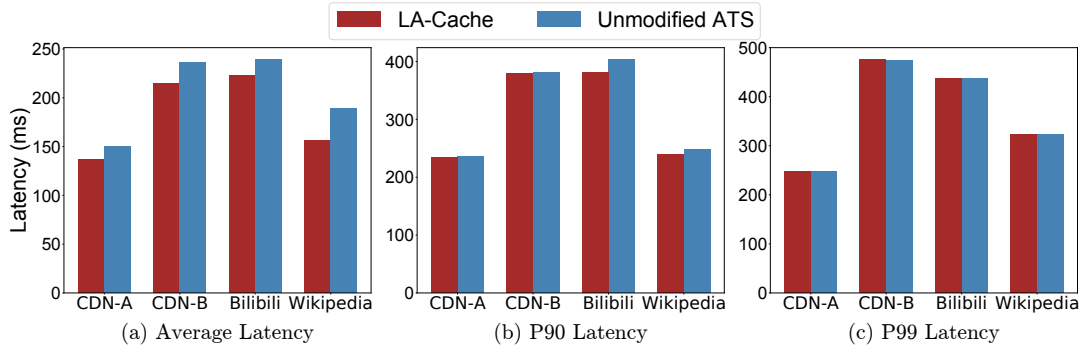


Figure 8: The latencies of LA-Cache and unmodified ATS using a 256GB cache.

		CDN-A		CDN-B		Bilibili		Wikipedia	
Metric	Experiment	LA-Cache	ATS	LA-Cache	ATS	LA-Cache	ATS	LA-Cache	ATS
Throughput (Gbps)	max	8.64	8.28	11.16	10.73	11.93	11.88	9.98	9.36
Overall CPU (%)	average	27.4	2.8	28.2	3.7	27.1	2.8	28.5	4.0
Peak Mem (GB)	max	2.6	2.3	2.7	2.5	2.3	2.2	2.3	2.2
P90 Latency (ms)	normal	234.2	235.6	378.8	381.6	390.7	403.7	239.0	247.3
P99 Latency (ms)	normal	247.6	248.2	474.1	474.6	435.5	436.5	323.7	324.0
Overall Latency (ms)	average	137.2	150.6	215.0	236.7	223.5	239.4	156.4	188.9

Table 3: Resource usage for LA-Cache and ATS in max (throughput-bound) and normal (production-speed) experiments.

6.2 Accuracy of Timer-based Model

We first show that our proposed timer-based model for delayed hits (see Sections 3 and 4) is accurate.

Non-Poisson arrivals in production traces. We first show that production traces do not have strict Poisson arrivals for any content. To that end, we generate a synthetic trace based on the real trace, where each content follows Poisson arrivals with the same average arrival rate as in the corresponding trace. We analyze the distribution of the inter-arrival times for the corresponding contents from the real and synthetic traces. It is clearly shown in Figure 6 that they are visibly different. Similarly trends hold for other contents in all production traces considered in this paper, i.e., production traces do not have strict Poisson arrivals for any content.

Comparison between theoretical and empirical results.

We now show that despite the fact that production traces may not be strictly Poisson, our proposed timer-based model with Poisson assumption works well in practice. To this end, we compare the theoretically computed average latency (calculated using Equation (3)) to the empirically computed latency. In particular, we compare three results: (i) *Theory*: theoretical latency for the trace; (ii) *E-real*: empirical latency for the trace; and (iii) *E-synthetic*: empirical latency for the synthetic Poisson trace with same content arrival rates as the trace as described earlier. Figure 7 compares the curves for all three cases in four production traces. We observe that the theoretical latency matches very well with the empirical latency for

the synthetic trace while the empirical latency for the real trace only differs slightly with the other two.

6.3 Latency Reduction of LA-Cache Prototype

We first compare our LA-Cache prototype to the ATS production systems in mean latency and implementation overhead as shown in Figure 8 and Table 3. The average RTT is 200 ms.

Latencies. Figure 8 compares the mean latency, the 90-th percentile latency (P90 latency) and the 99-th percentile latency (P99 latency) of LA-Cache and unmodified ATS using four production traces with a 256GB cache. LA-Cache consistently reduces the latency compared to ATS by 5%-20% on average¹⁰.

Implementation overhead. We then compare the implementation overhead of our LA-Cache prototype against unmodified ATS. We measure the throughput, CPU and memory utility under the “max” experiments, as shown in Table 3. We see that LA-Cache has no measurable throughput overhead but the peak CPU utilization increases to 27.4% from 2.8% for ATS under CDN-A, 28.2% from 3.7% for ATS under

¹⁰P90 (resp. P99) latency is the value of top 10% (resp. 1%) latency. Though P90 (resp. P99) latencies of LA-Cache are not significantly better than ATS, it only means that the largest latency values are similar. More importantly, it is obvious that the mean latency, a key metric for real system, of LA-Cache significantly outperforms ATS, i.e., LA-Cache improves mean latency greatly for most content requests.

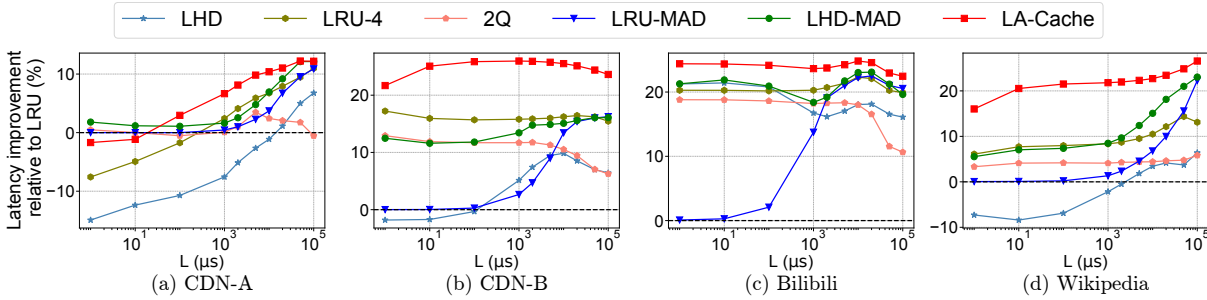


Figure 9: Comparison of mean latency improvement between LA-Cache and state-of-the-art algorithms relative to LRU using a 256GB cache.

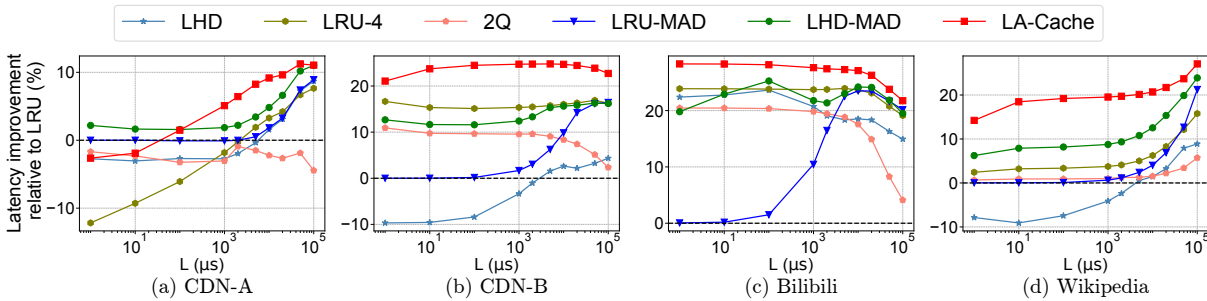


Figure 10: Comparison of mean latency improvement between LA-Cache and state-of-the-art algorithms relative to LRU using a 512GB cache.

CDN-B, 27.1% from 2.8% for ATS under Bilibili and 28.5% from 4.0% under Wikipedia. However, we note that most production servers, even at their busiest mode, have sufficient CPU headroom.

We replay our traces using its original timestamps and measure the latencies corresponding to cache misses, delayed hits and true hits. We call this “normal” experiments as shown in Table 3. It is clear that LA-Cache leads to significant latency reduction compared to ATS. More specifically, LA-Cache reduces the 90-th percentile latency (P90 latency) by 4%, the 99-th percentile latency (P99 latency) by 3%, and the overall average latency by 10% compared to ATS.

Finally, we measure the peak memory overhead for all traces and cache sizes, we observe that LA-Cache uses at most 1.1% of the cache size to store metadata. As we will show later, such a small loss in available caching space is more than offset by LA-Cache’s significant latency reduction.

From our above experiments in ATS, we believe that LA-Cache is a practical design for today’s CDNs and can be easily implemented in existing production CDN servers with modest resource overhead.

6.4 LA-Cache vs. State-of-the-art Algorithms

We further compare LA-Cache to a large number of state-of-the-art caching algorithms using four production traces with

a wide range of fetching latencies and cache sizes.

Latency. Figures 9 and 10 compare the mean latency improvement of LA-Cache and state-of-the-art caching algorithms with respect to LRU with different fetching latencies using a 256GB and a 512GB cache, respectively. We choose LRU as the baseline since major CDNs today still employ LRU or its variants for content caching. The comparisons with respect to the offline Bélády are relegated to the supplementary material for ease of readability.

Our LA-Cache consistently outperforms the best state-of-the-art algorithms, i.e., “the best-performing” algorithms in Table 1. Overall, LA-Cache reduces the latency by 5%-15% on average. Note that LA-Cache is robust across all traces in latency reduction whereas no existing state-of-the-art algorithms could robustly reduce the latency across all traces. In particular, LA-Cache outperforms LHD-MAD and LRU-MAD, two recently proposed latency-aware caching algorithms [4]. Our interpretation is that our LA-Cache naturally offers a variable fetching latency for different contents as well as fully captures the varying content sizes whereas LHD-MAD or LRU-MAD are designed under the assumption that contents are of equal size, which is not the case in production systems (see Figure 5).

Impact of cache size. We further characterize the impact of cache size on the latency reduction of LA-Cache compared to state-of-the-art algorithms. Based on the results above,

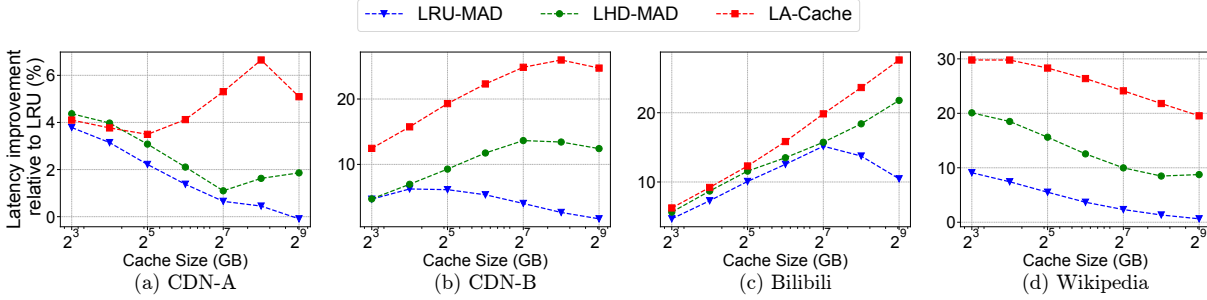


Figure 11: Comparison of the mean latency improvement between LA-Cache and state-of-the-art algorithms relative to LRU as a function of cache sizes.

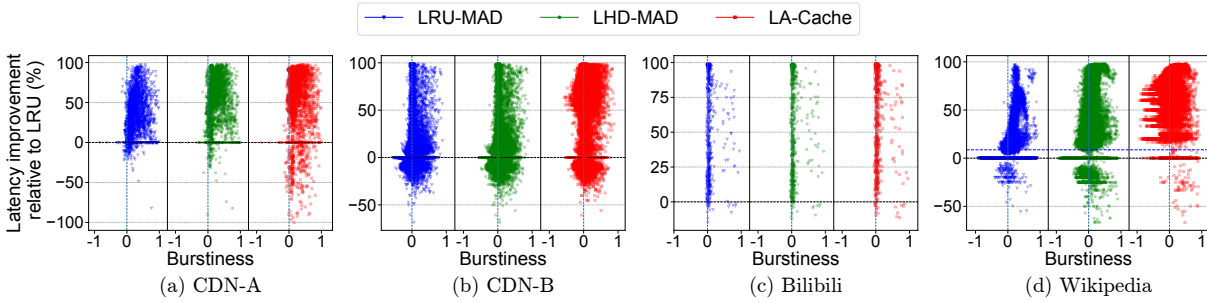


Figure 12: Improvement from the burstiness of content requests relative to LRU using a 256GB cache.

we only focus on the comparison with two latency-aware algorithms LRU-MAD and LHD-MAD. We compute the latency reduction compared to LRU for all traces while using a fixed value of $L=1$ ms. From Figure 11, we observe that LA-Cache’s improvement is between 4% and 30% compared to the widely deployed LRU. Finally, we see that LA-Cache outperforms LRU-MAD and LHD-MAD between 3% and 13% across different cache sizes.

Impact of burstiness. As motivated earlier in Section 2 as well as the design of our LA-Cache in Section 4, it is clear that a burst of requests to a content could contribute to the average latency more than a sparse of requests to a content. Now we turn to the question of *whether our intuition of burstiness indeed maps on the latency reduction of LA-Cache compared to state-of-the-art algorithms?*

To answer this question, we first need a measure to quantify the burstiness of a trace. A widely used metric is called the Goh-Barabasi score¹¹ [28]. However, it does not capture the impact of inter-arrival times, which play a significant role in delayed hits. To this end, we use a new burstiness measure [34] that not only captures the mean, variance of request sequences but also the inter-arrival times. The large the burstiness value, the busy the requests are. We refer interested readers to [34] for a detailed discussion¹².

¹¹The value of Goh-Barabasi score is between -1 to 1, with -1 being a regular request sequence, 0 being a random request sequence and 1 being a bursty request sequence.

¹²The Goh-Barabasi score is a statistical measure of burstiness in a se-

quence of events and is defined as $B = \frac{r-1}{r+1}$, where $r = \sigma/\mu$ is the coefficient of variation, σ and μ denote the standard deviation and the mean of inter-arrival times, respectively. However, the behavior of this score may not be robust with respect to finite-size request sequence. [34] redefined the burstiness score as $\frac{\sqrt{n+1}r - \sqrt{n-1}}{(\sqrt{n+1}-2)r + \sqrt{n-1}}$ where n is the sample size (i.e., number of requests in the sequence). This new score has been shown to quantify the burstiness in the empirical dataset without finite-size effects.

We characterize the burstiness of each content and the corresponding latency improvement of this content due to the latency-aware caching algorithms in Figures 12 and 13, where we consider the setting as above with a fixed value of $L=1$ ms. We observe that bursty contents (with a large burstiness value) incur a lower latency compared to LRU in general. Furthermore, we indeed observe that LA-Cache prioritizes more bursty requests compared to other state-of-the-art algorithms, which contributes to the latency reduction of LA-Cache. As a result, the overall mean latency is reduced. For example,

¹³We compute a weighted average burstiness score over all requests in the trace using the new burstiness measure [34], where the weight for each content is proportional to the total number of requests for this content.

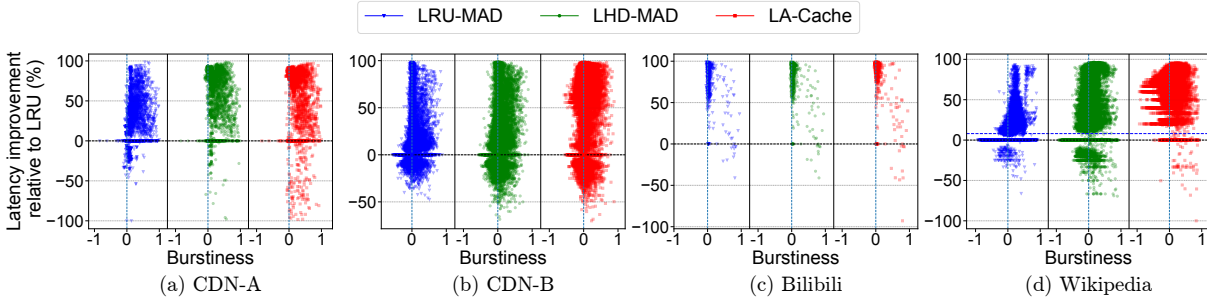


Figure 13: Improvement from the burstiness of content requests relative to LRU using a 512GB cache.

for Wiki in Figure 12 (d), it is obvious that LA-Cache prioritizes more bursty requests compared to LRU-MAD and LHD-MAD. This leads to a latency improvement of 22% for LA-Cache, while the improvements for LRU-MAD and LHD-MAD are 2% and 9%, respectively, as shown in Figure 9 ($L = 10^3 \mu s$). This phenomenon can also be observed when compared with the offline Bélády (see supplementary material). These observations further validate our intuitions on designing a ranking function to prioritize bursty contents so as to minimize latency (see Section 4).

7 Related Work

Caching algorithm design has been extensively studied over years. However, most of the previous works have focused on improving caching hit probabilities. We classify them by admission or eviction. The widely used admission algorithms include AdaptSize [11], TinyLFU [19] and SecondHit [40], and among others where static features such as content sizes are used for admission [1, 20]. A large number of works proposed eviction algorithms from classic Least Recently Used (LRU) [18], RANDOM, FIFO, to more sophisticated ones that are more difficult to implement in practice, e.g., LRU-K [47], LFU-DA [3, 51], GDSF [15], ARC [44], CAR [6] and among others, where recency, frequency or their combinations are usually used for eviction decision [7, 13, 30].

Recently, machine learning has been used for caching algorithm design. On the one hand, some focus on learning content popularities for content eviction via deep neural networks (DNNs), e.g., DeepCache [45], FNN-Cache [22], Pop-Cache [54] and PA-Cache [21] or by approximating or imitating offline optimal Bélády for content eviction, e.g., LFO [10], LRB [53]. On the other hand, some algorithms learn to decide whether or not to admit a content upon a request (i.e., *content admission*) via reinforcement learning (RL), e.g., RL-Cache [35], CACA [29], RL-Bélády [58] and among others [57, 60]. Again, most of these designs are focusing on improving cache hits rather than minimizing caching latency.

Closest to our work is [4, 42]. In particular, [42] provides a lower bound on the performance of caching policies when

delayed hits exist. [4] characterizes the impact of delayed hits and proposes an online approximation algorithm MAD based on a hard offline optimization problem. However, MAD fails to account for variable fetching latency and different content sizes, which are the cases in production CDNs and are both captured by our LA-Cache.

8 Conclusion

In this paper, we designed latency-aware caching in the presence of delayed hits, and proposed a novel timer-based mechanism to capture the impact of delayed hits which provably optimizes the mean caching latency. Furthermore, our model captured variable fetching latency and different content sizes, providing a theoretical basis for the understanding and design of latency-aware caching for content delivery in latency-sensitive systems. Using our timer-based model, we proposed a lightweight latency-aware caching algorithm LA-Cache. We implemented a LA-Cache prototype within Apache Traffic Sever. Using production traces, we showed that LA-Cache consistently outperformed state-of-the-art algorithms on latency reduction with modest resource overhead.

Acknowledgements

We would like to thank our anonymous reviewers for their valuable feedback. We would like to thank our shepherd's insightful comments that improved the quality of the paper immensely. This work was supported in part by the National Science Foundation (NSF) grants CRII-CNS-NeTS-2104880 and RINGS-2148309, and was supported in part by funds from OUSD R&E, NIST, and industry partners as specified in the Resilient & Intelligent NextG Systems (RINGS) program. This work was also supported in part by the U.S. Department of Energy's Office of Energy Efficiency and Renewable Energy (EERE) under the Solar Energy Technologies Office Award Number DE-EE0009341. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

- [1] Marc Abrams, Charles R Standridge, Ghaleb Abdulla, Edward A Fox, and Stephen Williams. Removal Policies in Network Caches for World-Wide Web Documents. In *Proc. of ACM SIGCOMM*, 1996.
- [2] Apache Traffic Server, 2020. <https://trafficserver.apache.org/>.
- [3] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating Content Management Techniques for Web Proxy Caches. *ACM SIGMETRICS Performance Evaluation Review*, 27(4):3–11, 2000.
- [4] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S Berger. Caching with Delayed Hits. In *Proc. of ACM SIGCOMM*, 2020.
- [5] F. Baccelli and P. Brémaud. *Elements of Queueing Theory: Palm Martingale Calculus and Stochastic Recurrences*, volume 26. Springer Science & Business Media, 2013.
- [6] Sorav Bansal and Dharmendra S Modha. CAR: Clock with Adaptive Replacement. In *Proc. of USENIX FAST*, 2004.
- [7] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *Proc. of USENIX NSDI*, 2018.
- [8] Laszlo A. Bélády. A Study of Replacement Algorithms for A Virtual-Storage Computer. *IBM Systems journal*, 5(2):78–101, 1966.
- [9] D. Berger, P. Gland, S. Singla, and F. Ciucu. Exact Analysis of TTL Cache Networks. *Performance Evaluation*, 79:2–23, 2014.
- [10] Daniel S Berger. Towards Lightweight and Robust Machine Learning for CDN Caching. In *Proc. of ACM HotNets*, 2018.
- [11] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network. In *Proc. of USENIX NSDI*, 2017.
- [12] Bilibili. <https://www.bilibili.com>.
- [13] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic Caching: Flexible Caching for Web Applications. In *Proc. of USENIX ATC*, 2017.
- [14] H. Che, Y. Tung, and Z. Wang. Hierarchical Web Caching Systems: Modeling, Design and Experimental Results. *IEEE Journal on Selected Areas in Communications*, 20(7):1305–1314, 2002.
- [15] Ludmila Cherkasova. *Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching policy*. Hewlett-Packard Laboratories, 1998.
- [16] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *Proc. of USENIX NSDI*, 2016.
- [17] Cisco. Cisco Visual Networking Index: Forecast and Trends 2022. *Cisco, Tech. Rep*, 2019.
- [18] Edward Grady Coffman and Peter J Denning. *Operating Systems Theory*. Prentice-Hall Englewood Cliffs, NJ, 1973.
- [19] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage*, 13(4):1–31, 2017.
- [20] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. of USENIX NSDI*, 2013.
- [21] Qilin Fan, Xiuhua Li, Jian Li, Qiang He, Kai Wang, and Junhao Wen. PA-Cache: Evolving Learning-Based Popularity-Aware Content Caching in Edge Networks. *IEEE Transactions on Network and Service Management*, 18(2):1746–1757, 2021.
- [22] Vladyslav Fedchenko, Giovanni Neglia, and Bruno Ribeiro. Feedforward Neural Networks for Caching: Enough or Too Much? *ACM SIGMETRICS Performance Evaluation Review*, 46(3):139–142, 2019.
- [23] A. Ferragut, I. Rodríguez, and F. Paganini. Optimizing TTL Caches under Heavy-tailed Demands. In *Proc. of ACM SIGMETRICS*, 2016.
- [24] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure Accelerated Networking: Smartnics in the Public Cloud. In *Proc. of USENIX NSDI*, 2018.
- [25] N. C. Fofack, M. Dehghan, D. Towsley, M. Badov, and D. L. Goeckel. On the Performance of General Cache Networks. In *VALUETOOLS*, 2014.
- [26] N. C. Fofack, P. Nain, G. Neglia, and D. Towsley. Analysis of TTL-based Cache Networks. In *VALUETOOLS*, 2012.
- [27] Davy Genbrugge and Lieven Eeckhout. Memory Data Flow Modeling in Statistical Simulation for the Efficient Exploration of Microprocessor Design Spaces. *IEEE Transactions on Computers*, 57(1):41–54, 2007.

- [28] K-I Goh and A-L Barabási. Burstiness and Memory in Complex Systems. *EPL (Europhysics Letters)*, 81(4):48002, 2008.
- [29] Yu Guan, Xinggong Zhang, and Zongming Guo. CACA: Learning-based Content-Aware Cache Admission for Video Content in Edge Caching. In *Proc. of ACM Multimedia*, 2019.
- [30] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *Proc. of USENIX ATC*, 2015.
- [31] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An Analysis of Facebook Photo Caching. In *Proc. of ACM SOSP*, 2013.
- [32] Stratis Ioannidis and Edmund Yeh. Adaptive Caching Networks with Optimality Guarantees. *Proc. of ACM SIGMETRICS*, 2016.
- [33] J. Jung, A. Berger, and H. Balakrishnan. Analysis of TTL-based Cache Networks. In *IEEE INFOCOM*, 2003.
- [34] Eun-Kyeong Kim and Hang-Hyun Jo. Measuring Burstiness for Finite Event Sequences. *Physical Review E*, 94(3):032311, 2016.
- [35] Vadim Kirilin, Aditya Sundarajan, Sergey Gorinsky, and Ramesh K Sitaraman. RL-Cache: Learning-based Cache Admission for Content Delivery. *IEEE Journal on Selected Areas in Communications*, 38(10):2372–2385, 2020.
- [36] Jian Li, Truong Khoa Phan, Wei Koong Chai, Daphne Tuncer, George Pavlou, David Griffin, and Miguel Rio. DR-Cache: Distributed Resilient Caching with Latency Guarantees. In *Proc. of IEEE INFOCOM*, 2018.
- [37] Jian Li, Srinivas Shakkottai, John CS Lui, and Vijay Subramanian. Accurate Learning or Fast Mixing? Dynamic Adaptability of Caching Algorithms. *IEEE Journal on Selected Areas in Communications*, 36(6):1314–1330, 2018.
- [38] Yuanyuan Li and Stratis Ioannidis. Universally Stable Cache Networks. In *Proc. of IEEE INFOCOM*, 2020.
- [39] limCacheSim. <https://github.com/1a1a1a/libCacheSim>.
- [40] Bruce M Maggs and Ramesh K Sitaraman. Algorithmic Nuggets in Content Delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, 2015.
- [41] Milad Mahdian, Armin Moharrer, Stratis Ioannidis, and Edmund Yeh. Kelly Cache Networks. In *Proc. of IEEE INFOCOM*, 2019.
- [42] Peter Manohar and Jalani Williams. Lower Bounds for Caching with Delayed Hits. *arXiv preprint arXiv:2006.00376*, 2020.
- [43] Valentina Martina, Michele Garetto, and Emilio Leonardi. A Unified Approach to The Performance Analysis of Caching Systems. In *Proc. of IEEE INFOCOM*, 2014.
- [44] Nimrod Megiddo and Dharmendra S Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proc. of USENIX FAST*, 2003.
- [45] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. DeepCache: A Deep Learning based Framework for Content Caching. In *Proc. of Workshop on Network Meets AI & ML*, 2018.
- [46] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling Memcache at Facebook. In *Proc. of USENIX NSDI*, 2013.
- [47] Elizabeth J O’neil, Patrick E O’neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. *ACM SIGMOD Record*, 22(2):297–306, 1993.
- [48] Nitish K Panigrahy, Jian Li, and Don Towsley. Hit Rate vs. Hit Probability based Cache Utility Maximization. *ACM SIGMETRICS Performance Evaluation Review*, 45(2):21–23, 2017.
- [49] Nitish K Panigrahy, Jian Li, Don Towsley, and Christopher V Hollot. Network Cache Design under Stationary Requests: Exact Analysis and Poisson Approximation. *Computer Networks*, 180:107379, 2020.
- [50] Nitish K Panigrahy, Jian Li, Faheem Zafari, Don Towsley, and Paul Yu. A TTL-based Approach for Content Placement in Edge Networks. In *EAI International Conference on Performance Evaluation Methodologies and Tools*, pages 1–21. Springer, 2021.
- [51] Ketan Shah, Anirban Mitra, and Dhruv Matani. An O(1) Algorithm for Implementing the LFU Cache Eviction Scheme. *no*, 1:1–8, 2010.
- [52] D Shasha and T Johnson. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. of VLDB*, 1994.
- [53] Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. Learning Relaxed Belady for Content Distribution Network Caching. In *Proc. of USENIX NSDI*, 2020.

- [54] Kalika Suksomboon, Saran Tarnoi, Yusheng Ji, Michihiro Koibuchi, Kensuke Fukuda, Shunji Abe, Nakamura Motonori, Michihiro Aoki, Shigeo Urushidani, and Shigeki Yamada. PopCache: Cache More or Less based on Content Popularity for Information-Centric Networking. In *Proc. of IEEE LCN*, 2013.
- [55] Edward S Tam. *Improving Cache Performance via Active Management*. University of Michigan, 1999.
- [56] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced Photo Caching on Flash for Facebook. In *Proc. of USENIX FAST*, 2015.
- [57] Haonan Wang, Hao He, Mohammad Alizadeh, and Hongzi Mao. Learning Caching Policies with Sub-sampling. In *NeurIPS Machine Learning for Systems Workshop*, 2019.
- [58] Gang Yan and Jian Li. RL-Bélády: A Unified Learning Framework for Content Caching. In *Proc. of ACM Multimedia*, 2020.
- [59] Gang Yan, Jian Li, and Don Towsley. Learning from Optimal Caching for Content Delivery. In *Proc. of ACM CoNEXT*, 2021.
- [60] Chen Zhong, M Cenk Gursoy, and Senem Velipasalar. A Deep Reinforcement Learning-based Framework for Content Caching. In *Proc. of IEEE CISS*, 2018.

A Supplementary Material

A.1 Analysis of Real Request Traces

In this subsection, we provide the detailed analysis of four production traces used in this paper, as discussed in Section 6.

The content popularity distribution (Figure 5(a)) shows that most traces follow approximately a Zipf distribution with the Zipf parameter α between 0.56 and 1.24.

The content inter-arrival time distribution (Figure 5(b)) - the distribution of the time between two consecutive request arrivals - further distinguishes these traces. It is clear that the requested contents in CDN-A have the largest variations since CDN-A contents have the smallest inter-arrival times, i.e., most contents are only requested over a shorter time period. In contrast, CDN-B, Bilibili and Wikipedia serve millions of different customers and hence exhibit largely independent requests with random inter-request times, which is consistent with the content popularity distribution.

A.2 Comparison with Offline Optimum Bélády

We also compare the performance in term of latency reduction to the offline optimum Bélády. From Figures 14 and 15,

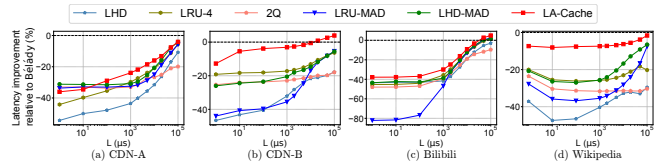


Figure 14: Comparison of the mean latency improvement between LA-Cache and state-of-the-art algorithms relative to Bélády using a 256GB cache.

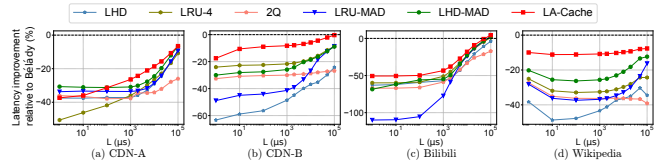


Figure 15: Comparison of the mean latency improvement between LA-Cache and state-of-the-art algorithms relative to Bélády using a 512GB cache.

we observe again that LA-Cache outperforms other state-of-the-art algorithms, and importantly LA-Cache can outperform Bélády as the fetching latency increases. For example, LA-Cache outperforms Bélády when L is greater than 0.8 ms in CDN-B, and when L is greater than 60 ms in Bilibili.

Finally, we characterize the impact of cache size when compared with the offline optimum Bélády. From Figure 16, we observe that LA-Cache consistently outperforms LRU-MAD and LHD-MAD across a wide range of cache sizes. More interestingly, LA-Cache outperforms Bélády.

Impact of burstiness. Complementary to the results presented in Figure 12, the burstiness of each content and the corresponding latency improvement of this content due to the latency-aware caching algorithms with respect to the offline Bélády with 256GB and 512GB cache are presented in Figure 17 and Figure 18, respectively. Again, we observe that LA-Cache prioritizes more bursty requests compared to other state-of-the-art algorithms, which contributes to the latency reduction of LA-Cache.

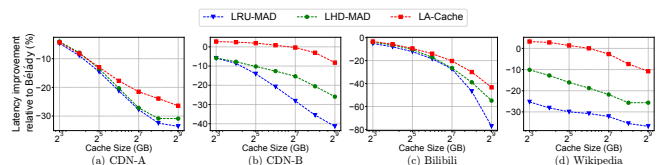


Figure 16: Comparison of the mean latency improvement between LA-Cache and state-of-the-art algorithms relative to Bélády as a function of cache sizes.

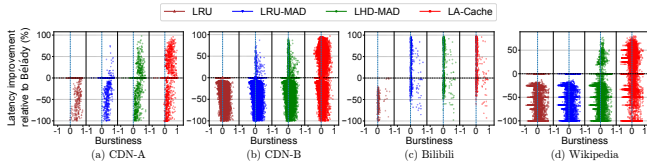


Figure 17: Improvement from the burstiness of content requests relative to the offline Bélády using a 256GB cache.

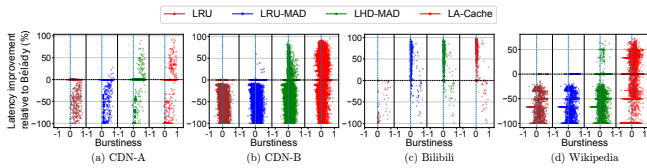


Figure 18: Improvement from the burstiness of content requests relative to the offline Bélády using a 512GB cache.

A.3 Hit Rate Comparison

Although we focus on designing latency-optimal caching in the presence of delayed hits, in which the goal of maximizing cache hits and the goal of minimizing latency are not equivalent (see Section 2.2), we argue that the latency improvements in turn also contribute to the cache hits performance. In most of existing works, the user-perceived latency upon cache hits is negligible, which is not true in the presence of “delayed hits” due to network latency. As a result, a content request results in three outcomes, i.e., miss, delayed hit, and true hit (see Introduction and Fig. 1). We observe that LA-Cache improves the true hit ratio up to 7% as shown in Figures 19 and 20, and “all hits” (true hit plus delayed hit) up to 9% as shown in Figures 21 and 22.

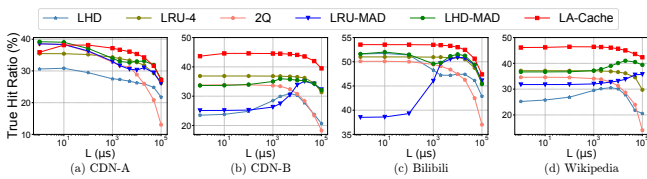


Figure 19: Comparison of true hits between LA-Cache and state-of-the-art algorithms using a 256GB cache.

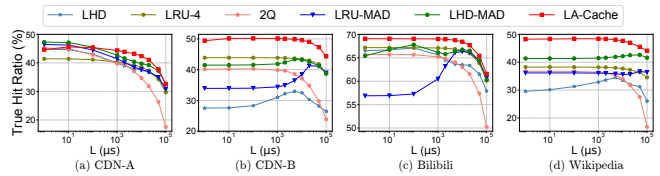


Figure 20: Comparison of true hits between LA-Cache and state-of-the-art algorithms using a 512GB cache.

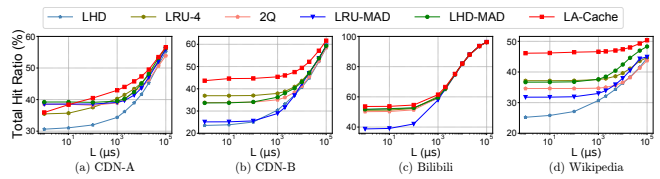


Figure 21: Comparison of all hits between LA-Cache and state-of-the-art algorithms using a 256GB cache.

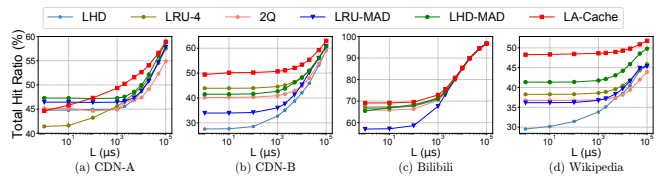


Figure 22: Comparison of all hits between LA-Cache and state-of-the-art algorithms using a 512GB cache.

Hashing Design in Modern Networks: Challenges and Mitigation Techniques

Yunhong Xu[†], Keqiang He[‡], Rui Wang[‡], Minlan Yu^{§‡}, Nick Duffield[†],
Hassan Wassel[‡], Shidong Zhang[‡], Leon Poutievski[‡], Junlan Zhou[‡], Amin Vahdat[‡]
[†]Texas A&M University [§]Harvard University [‡]Google, Inc.

Abstract

Traffic load balancing across multiple paths is a critical task for modern networks to reduce network congestion and improve network efficiency. Hashing which is the foundation of traffic load balancing still faces practical challenges. The key problem is there is a growing need for more hash functions because networks are getting larger with more switches, more stages, and increased path diversity. Meanwhile, topology and routing become more agile in order to efficiently serve traffic demands with stricter throughput and latency SLAs. On the other hand, current generation switch chips only provide a limited number of uncorrelated hash functions. We first demonstrate why the limited number of hashing functions is a practical challenge in today's datacenter network (DCN) and wide-area network (WAN) designs. Then, to mitigate the problem, we propose a novel approach named *color recombining* which enables hash functions to reuse via leveraging topology traits of multi-stage DCN networks. We also describe a novel framework based on *coprime* theory to mitigate hash correlation in generic mesh topologies (i.e., spineless DCN and WAN). Our evaluation using real network trace data and topologies demonstrate that we can reduce the extent of load imbalance (measured by the coefficient of variation) by an order of magnitude.

1 Introduction

Traffic load balancing is critical to the reliability and efficiency of modern datacenter and wide-area networks [16, 34, 37]. One widely deployed technique for traffic load balancing is Equal-Cost Multi-Path (ECMP) routing [16], where packets forwarding to a destination are load-balanced over multiple paths based on hashing of the packet header that takes place in switch hardware. ECMP and its variant Weighted-Cost Multi-Path (WCMP) [37] allow proper utilization across abundant paths available in modern networks. Hashing on header fields allows packets of the same flow to follow the same path without incurring packet reordering. As ECMP/WCMP offers

a number of nice properties, including stateless operation and no reordering, it is the de facto standard for traffic load balancing in large IP networks [1, 28, 37].

Switch hashing is the foundation of traffic load balancing in modern multi-path networks. To optimally load balance traffic and utilize full available bandwidth in multi-path networks using ECMP/WCMP, hash function allocation across a network should adhere to the following rule: *no correlated hash functions should appear on the same forwarding path in the absence of color recombining* (§3). Unfortunately, commodity switch chips only support a limited number of hash functions. For example, the software development kit for Broadcom switches supports *RTAG7* [9], a hashing scheme utilizing seven hash functions; the Cisco Nexus 5500 Series offers eight versions of *CRC8* [8]; the switches deployed in our datacenters have six independent hash functions. It is difficult to implement a large set of complex hash functions because hash computation becomes a bottleneck at high line rates [17, 19] and switch chip architects often need to trade-off between high line rate and hash function complexity.

Because of the limited number of hash functions provided in switch chips, a challenge in network design is that the reuse of correlated or even identical hash functions in different switches along the same end-to-end path for a flow causes *traffic polarization and inherent load imbalance*. Many network operators have observed this problem before [7, 11, 14, 18, 25]. One example is the Cisco Express Forwarding (CEF) Polarization phenomenon [7], where different switches repeatedly use the same hashing algorithm, resulting in a switch selecting a small portion of links for all traffic destined for one prefix, while other links were underutilized. Another example is the hashing imperfection problem observed by a large cloud provider [18, 25], where correlated hash functions lead to network congestion and even high priority traffic losses which directly impact application performance. In this paper, we use the term *hash correlation* to describe the association between hash functions in different switches that leads to traffic polarization.

Researchers and network operators have proposed a few

approaches to mitigate hash correlations. One class of methods uses variations of available hash functions. The most common approach uses hash functions with different seeds to avoid hash correlation. However, contrary to conventional wisdom, seeds are not as effective as expected as we will show theoretically and experimentally in this paper. Another approach uses Time-To-Live (TTL) in the packet header for hashing. There are two ways of using TTL: a) using TTL as part of hashing input, which has the following limitations in production: 1) it breaks traceroute because probe packets are hashed/pathed differently at the same hop; 2) IP in IP tunneling [12] (a common technique for network virtualization in the cloud) commonly uses the inner IP headers because they have more entropy than the outer IP headers, but the inner headers' TTL does not change. It's also hard to select outer header TTL and inner headers because switches do not have enough bit vectors. b) choosing a different hash function based on TTL [14], in addition to 1) and 2) mentioned in a), the challenges are: 3) it also requires modifying switch hardware, which presents administrative and commercial barriers to implementation since switch chips deployed in most datacenters do not support this operation currently; and 4) it is still constrained by the limited number of hash functions implemented in ASIC.

This paper focuses on fixed-function switches which are still the majority of devices in datacenters in the industry today. Hyperscalers cannot replace all switches with programmable ones in the near future. In this paper, we first demonstrate that the limited number of hash functions in commodity switch chips poses a practical challenge in modern DCNs and WANs and hinders the development of more advanced network topology and routing designs. To overcome this challenge, we propose two novel techniques that improve hashing, the cornerstone of traffic load balancing, substantially while respecting the limited number of independent hash functions available in commodity switches— 1) for widely adopted, multi-stage Clos DCNs, we propose a *color recombining* approach by comparing the effect of hash functions on traffic to light passing through multiple triangular prisms: the color recombining technique leverages the trait of multi-stage Clos topology where polarized traffic gets recombined to *color white* after passing through a multi-stage Clos. Color recombining enables us to reuse certain hash functions along the forwarding path without causing hash correlations. We discuss hashing design for one of the multi-stage Clos DCNs, Jupiter [26], and how the color recombining technique helps reduce the required number of independent hash functions; 2) for non-hierarchical mesh networks such as spineless DCN [13, 32] and WAN, we propose a novel framework residing in the Software Defined Networking (SDN) [20] controller. The framework mitigates the effect of hash correlations by the selection of the divisors n (n is also known as group size in ECMP/WCMP routing) used to map a flow's hash value h to the output port over which a packet is forwarded, i.e., $h\%n$.

Specifically, we establish that *when ECMP/WCMP group sizes at different switches are coprime, then the effects of underlying correlations between hash functions are reduced significantly*. We add a small amount of logic to the SDN controller to ensure the group sizes of switches with correlated hash functions are coprime. As a software-only approach, this coprime-based approach is compatible with commodity switch chips.

To summarize, the contributions of our work are as follows:

1. Details the hashing design in *multi-stage Clos DCN* and proposes a *color recombining* method to allow hash function reuse and to reduce the number of hash functions needed in multi-stage Clos DCNs;
2. Identifies an approach based on the *coprime* theory to mitigate hash correlations for generic *mesh networks* such as *spineless DCN* and *WAN* that require a large number of independent hash functions, and proposes algorithms to coprime group sizes for both ECMP and WCMP routing;
3. Evaluation results based on real network trace data and topologies demonstrate color recombining and coprime techniques' effectiveness in mitigating hash correlation—they can reduce the extent of load imbalance (measured by the coefficient of variation, CV) by approximately an order of magnitude.

The color-recombining approach uses topology and forwarding structures that are common in Clos networks and requires no hardware or software changes except hash function reconfiguration. The coprime-based method works with any topologies (e.g., mesh networks) but it requires controller software changes and only resolves polarization when the conditions described in §4 are met.

In addition to the technical contributions, we also would like to call for switch vendors' attention to offer better hashing support in future generations of chips to facilitate flexible network designs.

2 Background and Motivation

In this section, we first motivate why hashing is an important problem in modern DCNs and WANs. Then, we provide the background of ECMP and WCMP. Finally, we introduce the traffic polarization issue caused by hash correlation and reveal the fact that the current generation of switch chips only provides a limited number of independent hash functions which poses a practical challenge to traffic load balancing in modern networks.

The implications of bad hashing are twofold. First, bad hashing leads to *traffic polarization* that endangers reliability (due to reduced path diversity), wastes network bandwidth, cancels efficiency gains of traffic engineering, and inevitably

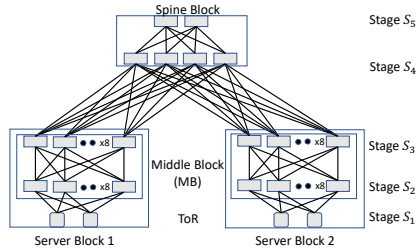


Figure 1: Illustration of Jupiter DCN (5-stage Clos). For simplicity, only one *Middle Block* of each *Server Block* and one *Spine Block* of the spine layer is shown.

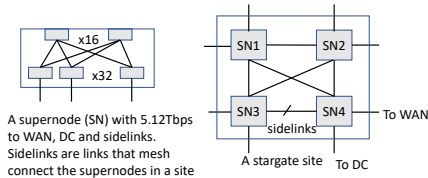


Figure 2: Illustration of a *Stargate* site in the mesh-connected B4 WAN. Figure is adapted from [15].

increases network cost. Second, bad hashing causes *inherent traffic load imbalance* and leads to network congestion that affects application performance.

2.1 Hashing is a Practical Challenge in Network Designs

There are several trends that make hashing an increasingly important problem in modern networks: 1) both datacenter and wide-area networks are getting bigger with more switches and stages; 2) modern networks are very dense and have a large number of paths between any two nodes; 3) topology and routing become more agile and flexible in order to improve network efficiency and availability, e.g., the move from spinefull DCN to reconfigurable spineless [13, 32] and the use of non-shortest path routing to improve availability and performance [15]; and 4) emerging applications (such as distributed machine learning) are becoming more throughput hungry while demanding stricter network SLA guarantees. These new trends pose challenges to commodity switches' hashing capability, which is the cornerstone of traffic load balancing.

2.1.1 Multi-stage Clos DCN

Modern DCN connects a massive amount of compute/storage nodes, runs critical services, such as Search Serving, Video Serving, Geo & Map, Cloud, and Gaming, etc, and has very large path diversity; therefore, hashing and traffic load balancing are crucial. We use the Jupiter topology [26, 35] (as illustrated in Figure 1) as the case study of multi-stage Clos DCNs. We denote a *Server Block* (aka Pod) as SB and the

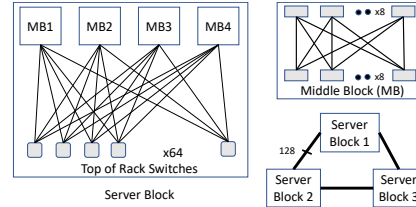


Figure 3: Spineless DCN [32] to simulate hash correlation's impact on traffic load balancing.

5 switch stages as S_1 to S_5 . So the longest forwarding path is when a packet is routed from a ToR in SB_i to another ToR in SB_j and the hop sequence is $S_1(SB_i) \rightarrow S_2(SB_i) \rightarrow S_3(SB_i) \rightarrow S_4 \rightarrow S_5 \rightarrow S_4 \rightarrow S_3(SB_j) \rightarrow S_2(SB_j) \rightarrow S_1(SB_j)$ where i, j are server block indices. To achieve optimal load balancing performance, hashing needs to follow the following property: no correlated hash functions should appear on the same forwarding path. Naively we need $O(2L)$ (more precisely, $max_number_of_hops - 1$) independent hash functions, where L is the number of layers (or stages) of the fabric. In the case of Jupiter, 8 hash functions are needed. Unfortunately, there are only 6 uncorrelated hash functions provided by the switch chips deployed in our datacenters and the requirement of 8 independent hash functions already makes the commodity switch's hashing capacity stretched. We will discuss how we reduce the number of hash functions needed for multi-stage Clos DCN in Section 3.

2.1.2 Spineless DCN

Recently, there are new *spineless* DCN topologies [13, 32] which reduce cost and enable faster tech refresh, but require more hash functions. Figure 3 is an example of spineless DCN topology. In spineless DCN, server blocks are directly connected via a mesh and the spine blocks are completely removed to reduce network cost (including the cost of both spine switches and the associated optics) significantly and to enable faster switch generation evolution. Also, non-shortest path routing is used to improve routing path diversity for high availability and enable traffic engineering to optimize network link utilization.

While spineless DCNs are cost-effective and efficient, hashing design becomes more challenging because the number of hash functions required depends on the number of server blocks instead of the number of layers as in multi-stage DCNs. Taking the Gemini [32] spineless DCN as an example, assuming that we only allow at most one transit server block in routing, then the longest forwarding path of a packet is: $S_1(SB_i) \rightarrow S_2(SB_i) \rightarrow S_3(SB_i) \rightarrow S_3(SB_j) \rightarrow S_2(SB_j) \rightarrow S_3(SB_j) \rightarrow S_3(SB_k) \rightarrow S_2(SB_k) \rightarrow S_1(SB_k)$ where i, j, k are the indices of three randomly chosen server blocks. The key challenge is that we need to make sure there are no correlated hash functions for any i, j, k combinations. So spineless DCN requires $O(N)$ hash functions where N is the number of server

blocks. N ranges from 10s to 100s in a typical fabric, so it is very challenging to design hashing with the limited hash functions provided by current-generation switch chips.

2.1.3 WAN

Traffic engineering and load balancing are critical to improving WAN's performance and reducing operational costs. Mesh-connected WANs have the same hashing design challenge where the number of uncorrelated hash functions required is subject to the scale of the network. For example, Figure 2 shows the topology of a Stargate site of the B4 WAN [15]. Each site is composed of up to 4 supernodes where a supernode is a 2-stage Clos with links to WAN, Data Center, and other supernodes in the same site. B4 site-level topology is a partially connected mesh and non-shortest path routing is employed for both availability and efficiency (i.e., traffic engineering) purposes. Similar to spineless DCNs, to ensure optimal load balancing performance, we need $O(N)$ hash functions, where N is the number of sites in the WAN. Note that B4 grew $7\times$ larger from 2012 to 2017 [15]. We will discuss the hash correlation mitigation technique for mesh networks such as spineless DCNs and WANs in Section 4.

2.2 ECMP/WCMP Traffic Load Balancing

Equal-Cost Multiple-Path (ECMP) [16] – a routing and traffic load balancing strategy that allows traffic between a source and destination node to be transmitted across multiple paths – identifies a set of routes, each of which is equal-cost towards the destination. The routes identified are referred to as an *ECMP group*. An ECMP group is defined at flow-level. When forwarding a packet, the routing strategy decides which next-hop path to use based on a hashing algorithm. That is, the route of a packet is determined by the mapping from the hash value to an egress port, i.e., $h \% n$, where h is the hash value, and n is the number of output ports in the ECMP group. The typical IP packet header fields used for hashing input are: source IP, destination IP, transport protocol, TCP/UDP ports, and IPv6 flow label.

Each route in an ECMP group has an equal chance for traffic forwarding. For example, Figure 4a shows the traffic from source IP prefix to destination IP prefix uses four equal-cost paths, which are labeled in four colors, where H_1 and H_2 are two independent hash functions.

When switch hashes packets across multiple paths according to customized weights instead of uniform ones, this variant of ECMP is called Weighted-Cost Multi-Pathing (WCMP) [37]. And the set of routes identified for a flow is referred to as a WCMP group. WCMP can be implemented via replicating ECMP table entries in the switch to approximate the intended WCMP weights. For example, if there are 2 output ports of a flow f , denoted by p_1, p_2 , and the intended weights are 2:1, then the WCMP group can be implemented

as p_1, p_1, p_2 (p_1 is duplicated as 2 entries to achieve the 2:1 traffic split).

2.3 Hash Correlation Causes Traffic Polarization and Load Imbalance

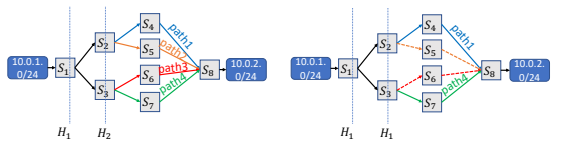
2.3.1 Limited Number of Hash Functions Leads to Hash Correlation

ECMP/WCMP is widely deployed in modern DCNs and WANs which have a large path diversity to improve traffic load balancing performance and reduce network congestion. Switches are configured with hash functions that compute hash values based on packet headers and forward packets via selecting one out of multiple next-hops based on the hash value and the ECMP/WCMP group size.

However, current-generation switch chips were designed for small-scale DCNs or large but sparse ISP networks and only provide a handful of independent hash functions. For example, the switch chips used in our datacenters provide six independent hash functions. The software development kit for Broadcom switches supports RTAG7 [9], a hashing scheme utilizing seven hash functions. The Cisco Nexus 5500 Series offers eight versions of CRC8 [8]. CRC and XOR are two popular hashing algorithms because these two algorithms have been used in communication systems with mature and efficient ASIC implementation, which includes plenty of circuit optimization [33]. As discussed in Section 2.1, the number of independent hash functions needed is far beyond (orders of magnitude difference) what is provided today, especially for spineless DCN and WAN.

In fact, it is difficult to implement a large set of uncorrelated hash functions because complex hash functions become a bottleneck at high line rates [17, 19]. For example, the authors in [19] discovered that the generation of Cyclic Redundancy Codes (CRCs) represents the main bottlenecks in iSCSI protocol processing. Switch chip architects often need to trade-off between high line rate and hash function complexity and we are not aware of any switch chips with cryptographic hash functions today due to computation complexity concerns.

In an ideal world where all the switches on the forwarding path are configured with completely independent hash functions, there would be no hash correlation. Due to the limited hash functions, we have to reuse them, which leads to hash correlation and traffic polarization. *Polarization* is a term used to describe what happens to light as it travels through a filter. Only light rays that have a certain characteristic get through the filter. We can take the same term and apply it to network traffic. Traffic polarization is the effect when a set of packets choose a particular path and the redundant paths remain completely unused [7]. Traffic polarization reduces path diversity and causes sub-optimal use of redundant paths and results in traffic load imbalance and network congestion. For example, Figure 4b shows that switches s_1, s_2 and s_3 employ the same



(a) Illustration example of ECMP. (b) ECMP with hash correlation.

Figure 4: ECMP with and without hash correlation. In (b), two out of four paths are unused due to hash correlation.

hash function (H_1). The hash correlation between switches s_2 (s_3) and s_1 causes traffic polarization. Figure 5 explains the problem: hash value on s_2 (s_3) is the same as s_1 , and thus flows are only forwarded to s_4 from s_2 and s_7 from s_3 . As a result, traffic from source to destination only uses two routing paths instead of four as in Figure 4a. Traffic polarization endangers reliability, wastes network capacity, and leads to hot links and network congestion under high traffic loads.

2.3.2 Random Seeds Are Not Effective

Since the shortage of independent hash functions available in switches prevents simply assigning independent hash functions to different switches, switch vendors suggest deriving multiple hash functions from one via using a switch-specific seed [7]: a seed is an initial value to start the CRC computation via XORing the input data. Because switch chip’s port count is powers of 2 and ports are typically divided into two equal-size directions (e.g., up-facing and down-facing) in modern networks, ECMP groups with an even number of ports are prevailing. However, we found that seeds do not work for an ECMP group of an even number of ports. For example, based on Theorem 1, all packets on switch s_2 (Figure 4b) from switch s_1 have the same routing choice even if they use different seeds for the same CRC; the proof of Theorem 1 can be found in the appendix.

Theorem 1 *If $crc_b(x \oplus z_1) \% 2 = crc_b(y \oplus z_1) \% 2$, $crc_b(x \oplus z_2) \% 2 = crc_b(y \oplus z_2) \% 2$, where x and y denote the data with the same size in bytes, z_1 and z_2 are two seeds of b bits, and b is the integer to denote the number of bits of the CRC, and \oplus denotes eXclusive OR (XOR).*

The above theorem indicates that choosing two random seeds does not create two independent hash functions from one CRC polynomial. Please note CRC polynomials are the most widely implemented hash functions in switch hardware, e.g., Broadcom’s RTAG7 is a hash family with 6 CRC16s and 1 CRC32. Applying a random seed is a linear operation and it can not decorrelate a hash function’s output effectively.

To confirm our theoretical analysis, we also build a spineless DCN topology (shown in Figure 3) to simulate traffic load balancing performance with hash functions provided

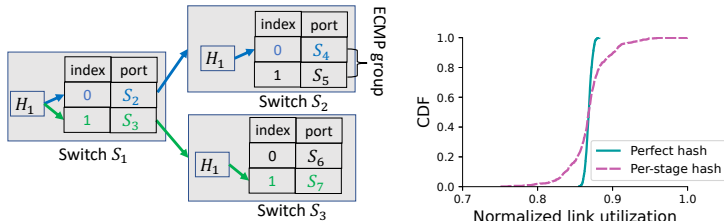


Figure 5: Hash correlation leads to correlated load balancing decisions. Figure 6: Normalized link utilization for perfect hashing and per-stage hashing

by a switch vendor. The topology is composed of three *Server Blocks* and each server block contains 64 ToRs (each ToR is assigned a /24 IP prefix). All these three server blocks have the same radix of 256. The three server blocks are connected in a full mesh (i.e., this is a spineless DCN [32]), so there are exactly 128 links between each server block pair. To simplify the analysis, we generate flows (the source IP and destination IP of a flow is randomly chosen from the source ToR and destination ToR’s IP prefix range respectively) following a uniform traffic pattern, i.e., all the server block pairs have the same amount of flows and the flow size distribution follows empirical datacenter flow size measurement in [4].

We compare two hashing schemes: one is *perfect hashing*¹ on each switch; and the other is *per-stage hashing with random seed* where the switches in different stages are configured with different functions provided by the switch vendor and each switch in the network is provided a completely independent and random seed. We measure the amount of traffic landed on the links between server blocks and show the normalized link utilization distribution in Figure 6. We can see from Figure 6 that the link utilization is close to uniform when using perfect hashing (which is expected) while per-stage hashing with random seeds leads to considerable traffic imbalance (max/min link utilization is 1.33). This experiment confirms that random seed does not solve the hash correlation problem.

3 Hashing Design in Multi-stage Networks

In this section, we describe the hashing design for multi-stage Clos DCNs. We first start with a strawman solution, *per-stage hashing with random seed*, then introduce *per-port hashing* scheme which requires more hashing functions than what is provided by commodity switches. In order to reduce the number of independent hash functions needed, we propose a novel approach named *color recombining* by leveraging topology traits of multi-stage Clos networks where polarized traffic is recombined into non polarized traffic and hash function reuse is allowed. We use Jupiter topology [26] as the case study here but the techniques proposed in this section can

¹Using the Mersenne Twister pseudo-random number generator in C++.

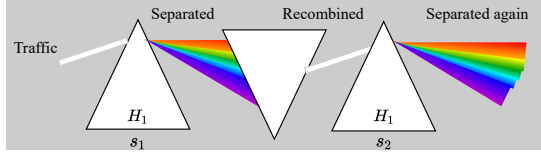


Figure 7: Illustration of color recombining concept.

be generalized to other multi-stage Clos DCNs. At the end of this section, we will provide details on why the hashing design for multi-stage Clos does not work in spineless DCNs and WANs.

One naive hashing design for Jupiter is *per-stage hashing with random seed* which means we assign a different hash function to each stage of switches and each switch is fed with a random seed. However, as we revealed in §2, random seed does not solve the hash correlation problem. Instead, we propose *per-port hashing* which means we apply hash function based on the input port of the switch. In total, there are 5 stages of switches in Jupiter as shown in Figure 1 and we apply two hash functions (one for upward traffic and the other for downward traffic) per switch except S_1 and S_5 . S_1 is ToR so only upward traffic requires a hash function; there is no need to have two hash functions in S_5 because we do not need to distinguish upward and downward traffic. Therefore, we need 8 independent hash functions to implement per-port hashing in Jupiter. And we can prove that for any routing path, there is no hash correlation. Taking the longest forwarding path as an example: a flow is routed from a ToR in the source server block to another ToR in the destination server block. So the forwarding path is S_1 (source ToR) $\rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_4 \rightarrow S_3 \rightarrow S_2 \rightarrow S_1$ (destination ToR). At each hop, a unique hash function is used so there is no hash correlation.

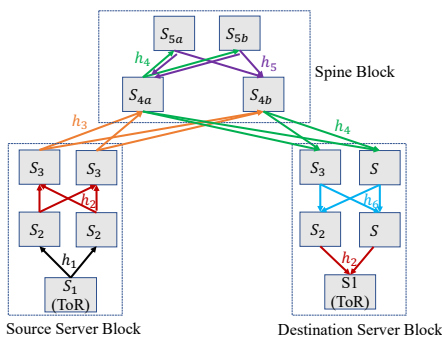


Figure 8: Per-port hashing with color recombining in Jupiter.

Per-port hashing design is clean and elegant (i.e., hash function allocation is static and easy to configure) for multi-stage Clos networks such as Jupiter, but the number of hash functions required is larger than what is provided by commodity switch chips. To reduce the number of hash function needed by per-port hashing, we identify and propose the *color recombining* technique via exploiting topology properties of

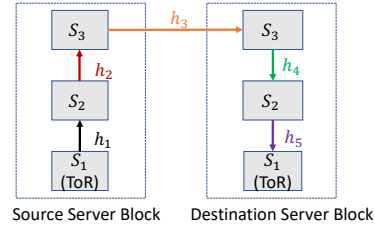


Figure 9: Per-port hashing works if only direct path (i.e., shortest path) routing is allowed in spineless DCN.

multi-stage Clos to enable hash function reuse. We compare traffic going through a hash function to light passing through a triangular prism: light passes through prism and gets separated to its component colors; analogously, network traffic goes through the hash function and gets forwarded to multiple paths. For example, Figure 7 shows that H_1 can be reused in switch s_2 because traffic becomes *color white* after passing through the middle triangular prism, where color white denotes the combined traffic before splitting or after recombining. We use the term *color recombining* to denote the recombining of polarized/dispersed traffic. Our key insight is that *in multi-stage Clos networks, certain stages of switches serve as the middle triangular prism in Figure 7 and combine polarized traffic to color white*.

As shown in Figure 8, there are two places traffic becomes "color white" in Jupiter – a) after passing the spine block and bouncing back to S_{4b} switches and b) after reaching the destination server block's S_2 switches. For a), h_4 is applied on S_{4a} towards the S_{5a} and S_{5b} direction. S_{5a} and S_{5b} apply an orthogonal hash function h_5 , and therefore, the same portion (50%) of $S_{4a} \rightarrow S_{5a}$ and $S_{4a} \rightarrow S_{5b}$ goes to S_{4b} . Effectively, the traffic of $S_{4a} \rightarrow S_{5a}$ and $S_{4a} \rightarrow S_{5b}$ towards S_{4b} recombines and h_4 can be reused. We define "polarized traffic w.r.t h_4 " as the unequally bucketized traffic through $S_{4a} \rightarrow S_{5a}$ and $S_{4a} \rightarrow S_{5b}$ respectively due to h_4 . For b), traffic gets polarized w.r.t h_2 on S_2 , however, after traffic reaching destination server block's S_2 chips, each S_2 is designed to receive the same portion of polarized traffic w.r.t h_2 , therefore h_2 can be reused on S_2 for downward traffic hashing. The sufficient condition of color recombining is *there exists a "color recombining stage" such that each switch in this stage receives the same portion of polarized traffic w.r.t H_x and after this color recombining stage, H_x can be reused without incurring hash correlation*. With color recombining, we effectively reduce the number of independent hash functions needed in Jupiter from 8 to 6 and all of the switch chips we use can meet this number.

Per-port hashing with color recombining inherits the pros of per-port hashing while requiring less hash functions from commodity switches. However, we found that this scheme breaks for the emerging spineless DCN topology [32]. In spineless DCN, both direct paths and transit paths are employed to route traffic because network architects want to a) increase path diversity to improve availability and b) per-

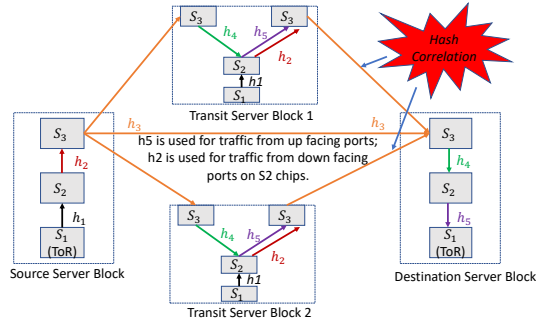


Figure 10: Per-port hashing breaks (h_3 appears twice on the forwarding path) in spineless DCN with non-shortest path routing.

form traffic engineering to hedge against unpredictable traffic spikes. Figure 9 shows that if we restrict routing to direct paths only, the per-port hashing scheme still works. However, as shown in Figure 10, we need to use h_3 twice for the flows traversing through a transit server block where the packet forwarding path is S_1 (source ToR) $\rightarrow S_2 \rightarrow S_3 \rightarrow S_3 \rightarrow S_2 \rightarrow S_3 \rightarrow S_3 \rightarrow S_2 \rightarrow S_1$ (destination ToR). Please note that traffic traversing through the transit server block bounces through S_2 chips for load balancing purposes. As mentioned in Section 2, we need $O(N)$ independent hash functions where N is the number of server blocks. N is a large number in our fabrics, so we need to identify more generic techniques to mitigate hash correlation. In Section 4, we will discuss the *coprime* technique for mesh topologies such as spineless DCN or WAN.

4 Mitigating Correlation for Mesh Networks

In this section, we describe a generic approach based on *coprime theory* to mitigate hash correlations in mesh networks. We first provide a theory about coprime in §4.1. Following it, we describe how the coprime theorem can be used to mitigate hash correlations for both ECMP and WCMP, in §4.2 and §4.3, respectively.

4.1 The Coprime Theorem

The key idea of the coprime theory is the modulo operation on coprime numbers (e.g., 127 and 128 are two coprime numbers) makes a hash function’s output uncorrelated. Considering one hash function H hashed to $\{0, 1, \dots, \hat{H}\}$, we propose to apply the modulo operation of two coprime values to derive two independent hash functions H_1 and H_2 from H , where \hat{H} is the highest hash value.

Below we explain how we use the coprime theory to mitigate hash correlation between two switches. In a switch, we use a hash function to choose the next hop via performing a modulo operation, i.e., $H(x)\%m$, where $H(x)$ is a hash value on a packet x , and m is the number of next hops in the ECMP

or WCMP group. Considering a scenario where two switches on a forwarding path both use H to choose the next hop, there exists hash correlation and traffic polarization as described in Figure 4b. Instead of using the same hash function H in these two switches, as denoted in Equation 1, we can use the derived H_1 ($H_1 = H\%q_1$) on the first switch to choose a next hop among m_1 next hops and H_2 ($H_2 = H\%q_2$) on the second switch, hashed to m_2 next hops. q_1 and q_2 are coprime numbers. The theorem 2 shows the two hash functions have no correlations.

$$H_i = H\%q_i, i \in \{1, 2\} \quad (1)$$

where q_1 and q_2 are two coprime values, and $q_i < \hat{H}$.

Theorem 2 $\forall i, \forall j, \Pr(H_2(x)\%m_2 = j | H_1(x)\%m_1 = i) \simeq \Pr(H_2(x)\%m_2 = j)$ if the following two conditions are satisfied:

Condition 1: $q_1 \gg m_1$ or $q_1\%m_1 = 0$, and $q_2 \gg m_2$ or $q_2\%m_2 = 0$;

Condition 2: $\hat{H} \gg q_1q_2$.

where q_1 and q_2 are two coprime values, m_1 and m_2 are the number of next hops, and x is a packet.

The theorem shows that the hash value of $H_2\%m_2$ is independent with the hash value of $H_1\%m_1$ for an input x when choosing proper coprimes q_1 and q_2 (q_1 and q_2 should be chosen to meet condition 1 and 2 of Theorem 2). The proof of Theorem 2 can be found in the appendix.

4.2 Coprime for ECMP

Based on Theorem 2, we propose a coprime-based approach to mitigate hash correlations along a routing path. When two hash functions are correlated, we just choose two coprimes and apply an extra modulo operation to derive two independent hash functions as in Equation 1.

While it may seem to be intuitive and easy to add an extra modulo operation to the hash value in a switch, but this requires switch hardware modification and no switch chips we are using provide this functionality. Even if switch vendors provide this functionality in their next generation chips, we have to replace all our existing switches, which is daunting and costly.

Instead, we propose to duplicate the ECMP group entries to match the coprime value in the SDN controller and interact with the switch flow and group tables via the existing OpenFlow [21] interface. Figure 11 shows the procedure of the method. We explain this using an example. Assuming that there are two egress ports in the ECMP group for IP prefix 10.1.2.0/24. Suppose one hash correlation occurs, we choose a coprime value of 5 to mitigate the hash correlation. Instead of modifying switch hardware to achieve two modulo operations $h\%5\%2$ (h is a hash value returned from the hash function) to choose an egress port, we duplicate 2 physical

egress ports into 5 logical ports in the ECMP group which are mapped to 2 physical egress ports. In this way, we only need one modulo operation, that is, $h\%5$, to determine the logical port and finally the physical egress port. Duplicating ECMP group entries to match the coprime value is supported by today's commodity switches and we only need to add a small amount of logic into the SDN controller. Note that Figure 11 shows that one flow table contains many IP prefixes/flows sharing the same group (multi-path) table in the switch, so we need to use an offset added to $hash\ value\ \%group\ size$ to index each IP prefix/flow.

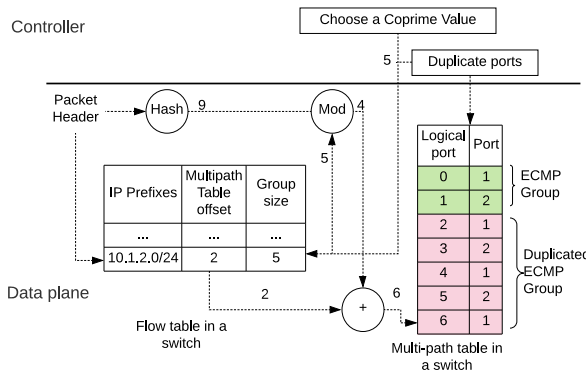
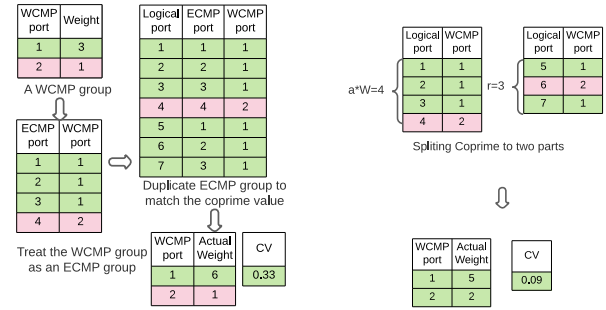


Figure 11: The procedure of applying a coprime number to mitigate hash correlation without requiring an extra modulo operation.

There are two technical challenges of the *coprime ECMP group size* method mentioned above: 1) increased switch memory usage, especially for large coprime values. The memory usage is increased by $O(q/m)$ times, where q and m is the coprime and original ECMP group size, respectively; and 2) ECMP precision loss – i.e., the difference between intended weights and actual weights of different egress ports of an ECMP group. In order to save switch memory, we may want to choose small coprime values. However, small coprime value contradicts condition 1 in Theorem 2; furthermore, we also need to tolerate the ECMP precision loss introduced by a small coprime value. When coprime ECMP group size, some ports are duplicated for $\lfloor q/m \rfloor$ times, and others are duplicated for $\lfloor q/m \rfloor + 1$ times. So when q/m is small, this introduces ECMP precision loss.

We use the coefficient of variation (CV) to measure how effective a coprime value is. Suppose we have m links, and the expected portion of traffic is $p_i = 1/m$ per link. After using coprime q , the actual traffic distribution is $\hat{p}_i = (\lfloor q/m \rfloor + I(i \leq q\%m))/q$, where $I(\cdot)$ is an indicator function, $i \in \{0, m-1\}$ is the port id, and when $i \leq q\%m$, $I(\cdot) = 1$. The CV is computed for \hat{p}_i . In our implementation of the coprime method, we minimize CV while obeying the ECMP table size limit offered by switch chips.



(a) Naive coprime: the WCMP group is treated as an ECMP group value 7 is split into the first $\hat{q} = a * W = 4$ and the remaining $r = 3$, where $a = \lfloor q/W \rfloor = 1$.

Figure 12: Illustration and comparison of the naive and improved algorithm to coprime WCMP groups.

4.3 Coprime for WCMP

As discussed in [37], topology asymmetry introduced by link or switch failures requires Weighted-Cost Multi-Path (WCMP) to distribute traffic in proportion to downstream hops' capacities. In this section, we extend the coprime-based method from ECMP to WCMP.

One straightforward method, denoted by *naive coprime*, is to treat a WCMP group as an ECMP group of W ECMP ports, where W is the sum of the weights, i.e., $W = \sum_i w_i$ and w_i is the weight of port i . We duplicate the W ECMP ports to q logical ports just as an ECMP group, where q is a coprime value. However, the weights after duplication could deviate significantly from the intended WCMP weights. One example is shown in Figure 12a: there are two ports in the WCMP group, and their weights are 3 and 1, that is, $w_1 = 3$ and $w_2 = 1$. This WCMP group can be treated as an ECMP group with $W = 4$ ECMP ports. Suppose, we choose a coprime value 7, and after duplication, the actual weights are $\hat{w}_1 = 6$ and $\hat{w}_2 = 1$, which are significantly different from the intended WCMP weights. We use the CV of $\{\hat{w}_i/w_i\}$ to quantify the difference. As shown in Figure 12a, the CV is 0.33.

To reduce the difference between the actual weights after coprime WCMP group size and the expected WCMP weights, we propose an improved algorithm to duplicate entries in the WCMP group, denoted by *split coprime*. Suppose the WCMP group has m ports, the weight is w_i for port i , $W = \sum_i w_i$, and the chosen coprime value is q . We split the coprime value to two parts: $\hat{q} = a * W$ and $r = q\%W$, where $a = \lfloor q/W \rfloor$. It is intuitive to duplicate the ports to \hat{q} logical ports, that is, each WCMP port are duplicated for exactly $w_i * a$ times. We duplicate m ports to left r entries in the following manner: each port i is replicated for $\lfloor r/m \rfloor + I(i < r\%m)$ times, where $I(\cdot)$ is an indicator function, and when $i < r\%m$, $i \in \{0, 1, \dots, m-1\}$, $I(\cdot) = 1$. Figure 12b illustrates the procedure of co-priming a WCMP group: the

coprime value $q = 7$ is split into $\hat{q} = 4$ and $r = 3$; for \hat{q} , the two WCMP ports are duplicated for w_i times, and $w_1 = 3$ and $w_2 = 1$; For r , port 1 is replicated for two times and port 1 for once. With this improved algorithm, the CV is much smaller than the naive way of simply treating a WCMP group as an ECMP group.

For WCMP, the memory cost of coprime is negligible because even without coprime, we need to do WCMP quantization (i.e., approximating fractional weights via duplicating ECMP table entries) [37] to ensure the number of WCMP entries in a group doesn't exceed a pre-defined limit.

5 Evaluation

We conducted simulations using three types of network topologies and real-world traffic traces. We also evaluated on a hardware testbed with hundreds of switches and large-scale production fabrics.

5.1 Experiment Setup

Traffic traces We use CAIDA trace dataset [5] from a high-speed Chicago monitor on a commercial backbone link: each 1-second segment has 475.37k packets and 12.91k 5-tuple flows on average. Hashing is the foundation of traffic load balancing in multi-path networks and it is applied at the flow-level in ECMP/WCMP routing, therefore we focus on flow-level statistics rather than packet-level in our simulation to quantify the goodness of a hashing design. To map a traffic trace to a network topology we evaluate, we randomly assign each IP in the traffic trace to one host in our topology.

Network topologies We employ three types of topologies in our evaluation (Table 1): 1) the simplified multi-stage Clos (Jupiter) DCN topology as depicted in Figure 1; 2) a spineless DCN which has eight server blocks and the connection among server blocks is DRing [13]. Each server block contains 8 S_3 chips and 8 S_2 chips (each chip has 16 ports). Every server block is directly connected to 4 neighboring server blocks and each server block pair has 16 direct links; 3) two ISP topologies [27]. The routing strategy for the ISPs is k -shortest ($k = 4$) path routing [31] in our simulation.

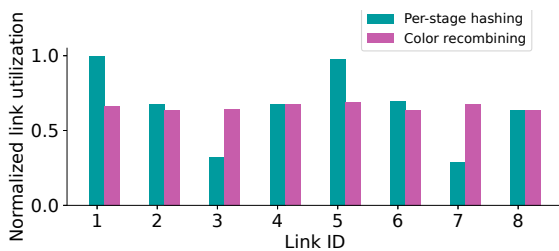


Figure 13: The normalized link utilization of eight links from one ECMP group for color recombining and per-stage hashing.

Topology	#Nodes	#Links
Spinefull DCN (Figure 1)	2 server blocks	128
Spineless DCN (DRing [13])	8 server blocks	512
ISP1 (small)	69	146
ISP2 (large)	122	371

Table 1: The topologies used in our simulation.

Hash functions We use one of the most widely used hash function family, RTAG7 [10], which includes seven hash functions (6 CRC16s and 1 CRC32) in our simulation.

Metrics We employ the Coefficient of Variation (CV), which is defined as δ/μ (δ is the standard deviation and μ is the mean of a set of data points), to quantify the goodness of hashing. CV is a commonly used statistical measure of the dispersion of data points around the mean. For an ECMP group with m output ports and the associated link utilization set $\{x_i\}$, $1 \leq i \leq m$, CV is computed against set $\{x_i\}$ directly. For a WCMP group with m output ports whose weights are w_i , $1 \leq i \leq m$, CV is computed against set $\{x_i/w_i\}$.

5.2 Color Recombining for Multi-stage DCN

We first study the traffic load-balancing performance of the proposed per-port hashing with color recombining (denoted as *color recombining* below) scheme for multi-stage DCN and compare with per-stage hashing with random seed (denoted as *per-stage hashing*). For per-stage hashing, we assign an independent hash function for each stage and initialize each hash function with a random seed.

We present normalized link utilization of eight links from one ECMP group in Figure 13. It shows that all eight links have similar link utilization of around 0.67 for color recombining, but the per-stage hashing approach shows severe non-uniformity. In color recombining, we only reuse hash functions when color recombining happens, which eliminates hash correlation; but in per-stage hashing, certain hash functions are reused without considering correlations and random seeds are linear operations that can not decorrelate the reused hash functions.

We also compute the CV for each ECMP group and draw the CDF of CVs in Figure 14. All the CVs are under 0.05 for color recombining, but per-stage hashing's CV can be above 0.6 (note that a CV of 1 means the standard deviation is equal to the mean). In other words, per-port hashing with the color recombining approach reduces CV by approximately an order of magnitude compared with per-stage hashing with random seeds. Large CV value means links in the same ECMP group are not properly utilized and results in wasted network capacity and unnecessary hot links that lead to network congestion under heavy traffic load.

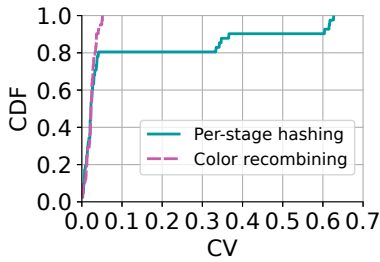


Figure 14: The CDF plot of CVs of each ECMP group. Compare color recombining with per-stage hashing.

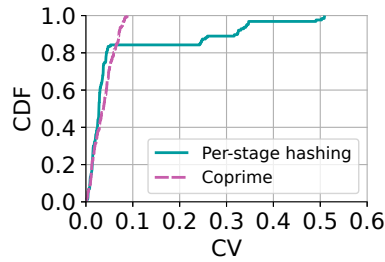


Figure 15: The CDF plot of CVs of each ECMP group. Compare coprime with per-stage hashing.

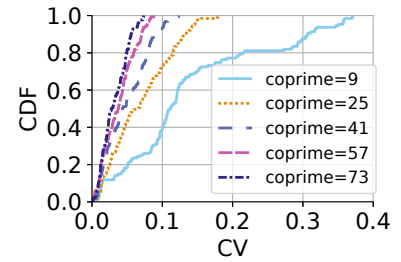


Figure 16: The CDF plot of CVs of each ECMP group. Compare different coprime values.

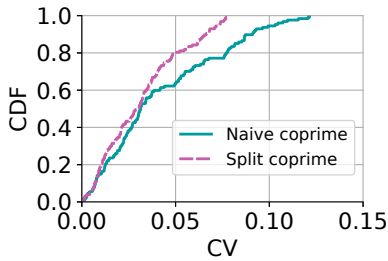


Figure 17: The CDF plot of CVs for each WCMP group. Compare two algorithms of coprime WCMP group.

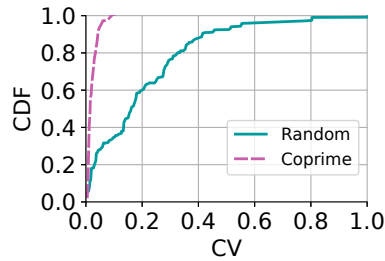


Figure 18: CDF plot of CV of each ECMP group for ISP 1.

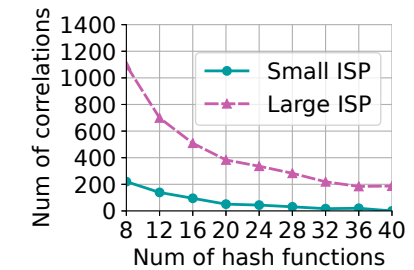


Figure 19: The number of ECMP groups of CV > 0.1.

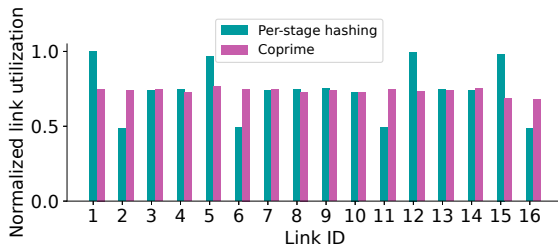


Figure 20: The normalized link utilization of 16 links connecting two server blocks.

5.3 Coprime for Spineless DCN

In the spineless DCN, all eight server blocks are connected in a DRing topology [13]. We choose two coprime values for every server block pair to mitigate the hash correlation between them. We compare coprime with the per-stage hashing where each server block uses 3 randomly chosen hash functions from the RTAG7 hash family and each function is supplied a random seed. We conduct experiments to evaluate the coprime-based approach for both ECMP and WCMP.

5.3.1 Coprime for ECMP

We show the normalized link utilization from 16 links connecting two server blocks in Figure 20. All links have the utilization of around 0.75 for the coprime-based approach,

where the coprime values are 8 (note each ECMP group has 8 ports because each S_3 chip of a server block has 8 up-facing ports) and 57 for the two server blocks, respectively. However, using per-stage hashing, the max/min link utilization ratio is above 2. Due to the shortage of independent hash functions, per-stage hashing has to reuse certain identical hash functions (even though they are provided with random seeds), and this raises traffic polarization as shown in Figure 20.

For quantification, the coprime-based approach outperforms per-stage hashing by reducing the CV by about 80%. For the coprime-based approach, all CVs are under 0.1, but for per-stage hashing, the CV can be as high as 0.5, as shown in Figure 15.

The coprime value matters when mitigating hash correlation: a large coprime value is more effective than a small one. We evaluate five different coprime values from 9 to 73, and show the CVs in Figure 16. It shows that the CV can be close to 0.4 when the coprime value is 9. When increasing the coprime value to 73, the improvement is not significant compared to 57. The result is consistent with our analysis in the end of § 4.2, which describes a trade-off between memory usage and ECMP precision. How to choose a coprime value depends on the memory a switch has and the level of imbalance one can tolerate.

5.3.2 Coprime for WCMP

To evaluate coprime for WCMP, we simulate a scenario where the first half links in an ECMP group have $2\times$ capacity, and therefore, the weights for those links are 2 and the remaining ones are 1. In this evaluation, we focus on two ways to duplicate the WCMP ports to match the coprime value: one is the *naive coprime* where a WCMP group is regarded as an ECMP as shown in Figure 12a; the other one (denoted as *split coprime*) is to split the coprime value into two parts as described in Figure 12b.

We draw the CDF plot of CVs per WCMP group in Figure 17. It shows that all CVs are under 0.075 when using the split coprime algorithm, while CV can reach 0.12 when using the naive coprime algorithm. We also observe split coprime reduces CV by approximately 60% for some WCMP groups compared with naive coprime. Our results indicate that split coprime should be preferred over naive coprime because it reduces CV using the same amount of switch memory.

5.4 Coprime for WAN

The coprime-based approach works for WAN topologies. We use the two ISP networks in Table 1 to evaluate load balancing performance of the coprime technique. We compare the coprime method with a random hash allocation approach (denoted as *Random* in Figure 18), where each switch randomly chooses an independent hash function from RTAG7 hash family and applies a random seed.

Our result shows that the coprime-based approach reduces CVs by about one order of magnitude compared with the random method. We compute all CVs for the link utilizations in every ECMP group and draw the CDF plot of CVs in Figure 18. When using the random method, CV can be as large as 1, which means only one link in the ECMP group is used, and others are idle. On the other hand, applying a coprime value to the ECMP group increases load balancing performance significantly, with CV lower than 0.1.

We also study how many independent hash functions are enough to eliminate traffic imbalance. We conduct an experiment by increasing the number of hash functions and gather the number of ECMP groups of $CV > 0.1$, caused by hash correlations. For this experiment, we introduce more CRC hash functions [24] beyond RTAG7, without considering the hardware implementation limitation. The result is shown in Figure 19. It shows that the small ISP (ISP 1) requires at least 40 hash functions to eliminate the correlations. However, the larger ISP (ISP 2) still has nearly 200 correlations when using 40 hashes. This result implies that even if chip vendors provide more hash functions in their next generation chips, it might be not sufficient for a large topology, e.g., B4 WAN grew $7\times$ larger in 5 years [15]. On the other hand, the coprime-base algorithm can be commonly used for topologies of arbitrary sizes.

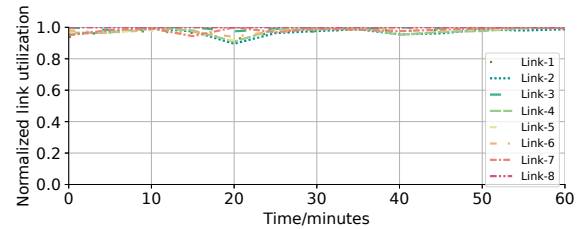


Figure 21: The normalized link utilization of one S_3 switch over 1-hour time window from the hardware testbed.

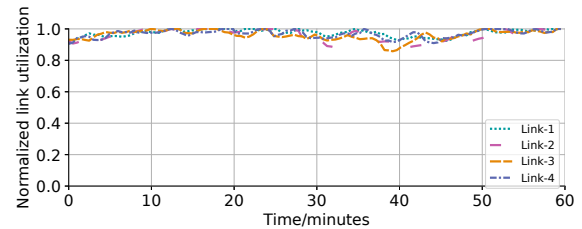


Figure 22: The normalized link utilization of four links from one spine block towards one server block over 1-hour time window from the production fabric after applying color recombining.

5.5 Hardware Testbed Evaluation and Production Fabric Deployment

We constructed a hardware testbed using the spineless DCN topology (Figure 3). There are 4 server blocks and hundreds of switches in this hardware testbed. The testbed serves a few Tbps traffic generated by production-grade applications and is carried by TCP and UDP. Within each server block, we apply a per-port hashing scheme as Figure 9 shows and in total 5 independent hash functions are used. We allow both shortest path routing and non-shortest path routing, i.e., we allow traffic to transit through an intermediate server block to reach a destination server block. To mitigate the hash correlation problem as illustrated in Figure 10, we apply the coprime scheme to coprime ECMP group size on S_3 switches – for traffic originating a server block, ECMP group size is coprime to 128; for traffic transiting a server block, ECMP group size is coprime to 127. Therefore, we can ensure there is no hash correlation for both shortest and non-shortest routing paths. We show the normalized link utilization of one S_3 switch over 1-hour time window in Figure 21. We observe excellent load balancing performance with a CV of 0.02.

Color-recombining has been deployed in our production multi-stage Clos fabrics for several years and significantly reduced load imbalance due to hash correlation. The deployment of color-recombining is the following: after deciding which hash functions can be reused, we simply configure the switches with the designated hash functions. We studied the hashing performance result of color-recombining in pro-

duction multi-stage Clos fabrics and the normalized link utilization of one representative spine block towards one server block from S_4 to S_3 direction is shown in Figure 22. The spine block's port speed is 40Gbps. As we discussed in Section 3 (Figure 8), hash function h_4 is reused to load balance traffic leaving spine blocks from S_4 to S_3 direction. But due to the color-recombining technique, polarized traffic due to h_4 are merged into color white when leaving S_4 chips, as a result, we observe very nice load balancing performance with a CV of 0.03.

6 Related Work

Traffic Load Balancing There are many prior works to address an important limitation of ECMP/WCMP: load balancing performance degrades when there is a large traffic entropy, i.e., when elephant flows collide on the same path, network congestion arises. For example, previous works [3, 28] propose to split elephant flows into smaller "flowlets" that can be load-balanced over different paths; other works [2, 30] propose to reschedule elephant flows after detecting collisions. MPTCP [29] is a transport protocol that uses subflows to transmit over multiple paths. These works rely on the assumption that there is no hash correlation. Our work is complementary and all load balancing schemes benefit from our improved hashing design, which is the corner stone of load balancing.

Hashing in Networks The universal algorithm [7] adds a 32-bit router-specific value to the hash function; however, as we show theoretically and experimentally in §2.3, random seeds do not work well. The paper [36] proposes to mitigate hash function correlation by randomly setting the VLAN-id for each switch. However, randomizing the VLAN-id increases network management complexity. The paper [14] selects a different hash function for a different value of the TTL field. However, this approach is still constrained by the limited number of hash functions and it also requires modifying switch hardware. The novelty of this paper is that our approaches work with commodity switch hardware without any hardware modification or switch upgrade, which is costly and daunting in large-scale networks.

Decorrelate Hashing Researches have already relied on prime numbers to design independent hash functions. For example, the universal hash functions employs the prime number as the divisor [6, 23], where primes are special case of coprimes. Our work extends prior theory and applies it to improve traffic load balancing in modern networks.

The patent by A. Meyer [22] uses co-primes to solve the storage collision for hash tables, that is, how to insert an item into the hash table when collision occurs; however, their method is different from ours where they add a co-prime offset to the original hash output, and this is similar to choosing an initial value for the hash function. We have proved that random initial values (including co-prime ones) do not work for an ECMP group of even size in Theorem 1.

7 Conclusions

This paper tackles a real but underestimated problem in network traffic load balancing, i.e., traffic polarization caused by hash correlations. This paper proposes two novel approaches to mitigate hash correlation: 1) a color recombining technique which exploits topology traits of Clos networks to allow hash function reuse and to reduce the number of needed independent hash functions in multi-stage Clos DCN and 2) a generic coprime-based technique to mitigate hash correlation for non-hierarchical mesh networks such as spineless DCN and WAN. Evaluations results based on real traffic trace and topologies show that the proposed techniques can reduce the extent of load imbalance, quantified by coefficient of variance (CV), by one order of magnitude. The limited hashing capability offered by current switch silicon reduces network reliability, efficiency and poses challenges to modern large-scale networks. We believe that novel approaches that work with current switch hardware are valuable. We also hope our use cases of more flexible topology and routing designs can motivate switch vendors to provide better hashing support in the future.

8 Acknowledgement

We thank our anonymous shepherd and reviewers for their helpful suggestions. The authors thank David Wetherall, Abdul Kabbani and Christophe Diot for their insightful feedback which greatly improves the quality of this paper. This work was supported in part by the National Science Foundation (grants CNS-1618030 and CNS-2107078).

References

- [1] Venkata Ramana Kiran Addanki. Method and system for management of flood traffic over multiple 0: N link aggregation groups, April 22 2014. US Patent 8,705,551.
- [2] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.
- [3] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 503–514. ACM, 2014.
- [4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center

- tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.
- [5] CAIDA. The CAIDA anonymized internet traces data access. http://www.caida.org/data/passive/passive_dataset_download.xml, 2019.
- [6] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.
- [7] Cisco. Cef polarization. <https://www.cisco.com/c/en/us/support/docs/ip/express-forwarding-cef/116376-technote-cef-00.html>, 2013.
- [8] Cisco. Data center access design with cisco nexus 5000 series switches and 2000 series fabric extenders and virtual portchannels. https://itnetworkingpros.files.wordpress.com/2014/04/c07-572829-01_design_n5k_n2k_vpc_dg.pdf, 2018.
- [9] Broadcom Corporation. Bcm56070 switch programming guide. <https://docs.broadcom.com/doc/56070-PG2-PUB>, 2020.
- [10] Dell. Dell configuration guide for the s4048-on system 9.9(0.0). https://www.dell.com/support/manuals/us/en/19/force10-s4048-on/s4048_on_9.9.0.0_config_pub-v1/rtag7, 2015.
- [11] Dell. Dell networking configuration guide for the mx1 10/40gbe switch i/o module 9.9(0.0). <http://www.dell.com/support/manuals>, 2015.
- [12] Network Working Group. Ip in ip tunneling. <https://datatracker.ietf.org/doc/html/rfc1853>.
- [13] Vipul Harsh, Sangeetha Abdu Jyothi, and P Brighten Godfrey. Spineless data centers. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 67–73, 2020.
- [14] Ariel Hendel. Mutable hash for network hash polarization, March 19 2015. US Patent App. 14/026,725.
- [15] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, et al. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google’s software-defined wan. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 74–87, 2018.
- [16] C. Hopps. Analysis of an equal-cost multi-path algorithm. RFC 2992, RFC Editor, November 2000.
- [17] Yuanhong Huo, Xiaoyang Li, Wei Wang, and Dake Liu. High performance table-based architecture for parallel crc calculation. In *The 21st IEEE International Workshop on Local and Metropolitan Area Networks*, pages 1–6. IEEE, 2015.
- [18] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.
- [19] Abhijeet Joglekar, Michael E Kounavis, and Frank L Berry. A scalable and high performance software iscsi implementation. In *FAST*, volume 5, pages 267–280, 2005.
- [20] Nick McKeown. Software-defined networking. *INFOCOM keynote talk*, 17(2):30–32, 2009.
- [21] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.
- [22] Alex Meyer. Co-prime hashing, July 28 2020. US Patent 10,725,990.
- [23] Mats Näslund. Universal hash functions & hard core bits. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 356–366. Springer, 1995.
- [24] Carnegie Mellon University Philip Koopman. Best crc polynomials. <https://users.ece.cmu.edu/~koopman/crc/>.
- [25] R. Wang, H. Wassel, J. Zhou, B. Felderman and D. Wetherall. Experiences with multipath forwarding in dc networks. 2017 Google Networking Research Summit.
- [26] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google’s datacenter network. In *ACM SIGCOMM computer communication review*, volume 45, pages 183–197. ACM, 2015.
- [27] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring isp topologies with rocketfuel. In *ACM SIGCOMM Computer Communication Review*, volume 32, pages 133–145. ACM, 2002.
- [28] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *NSDI*, pages 407–420, 2017.

- [29] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *NSDI*, volume 11, pages 8–8, 2011.
- [30] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: automating datacenter network failure mitigation. *ACM SIGCOMM Computer Communication Review*, 42(4):419–430, 2012.
- [31] Jin Y Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quarterly of Applied Mathematics*, 27(4):526–530, 1970.
- [32] Mingyang Zhang, Jianan Zhang, Rui Wang, Ramesh Govindan, Jeffrey C Mogul, and Amin Vahdat. Gemini: Practical reconfigurable datacenter networks with topology and traffic engineering. *arXiv preprint arXiv:2110.08374*, 2021.
- [33] Zhehui Zhang, Haiyang Zheng, Jiayao Hu, Xiangning Yu, Chenchen Qi, Xuemei Shi, and Guohui Wang. Hashing linearity enables relative path control in data centers. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 855–862, 2021.
- [34] Rui Zhang-Shen and Nick McKeown. Designing a predictable internet backbone with valiant load-balancing. *Quality of Service-IWQoS 2005*, pages 178–192, 2005.
- [35] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C Mogul, and Amin Vahdat. Minimal rewiring: Efficient live expansion for clos data center networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 221–234, 2019.
- [36] Junlan Zhou and Zhengrong Ji. Hashing technique to optimally balance load within switching networks. 2017.
- [37] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabani, Leon Poutievski, Arjun Singh, and Amin Vahdat. Wcmp: Weighted cost multipathing for improved fairness in data centers. In *Proceedings of the Ninth European Conference on Computer Systems*, page 5. ACM, 2014.

A Proofs of Theorems

Proof of Theorem 1 (CRC Seed) The CRC seed is also known as the initial value, where we can initialize the CRC

value via XORing the seed with the input byte. Suppose the input is denoted by x , and the two random seeds are z_1 and z_2 . In order to prove random seeds are not effective, we only need to prove the following two rules:

If $\text{crc}_b(x) \oplus \text{crc}_b(y) = \text{crc}_b(x \oplus z_1) \oplus \text{crc}_b(y \oplus z_1)$;

If $\text{crc}_b(x) \oplus \text{crc}_b(y) = \text{crc}_b(x \oplus z_2) \oplus \text{crc}_b(y \oplus z_2)$.

Let $z_1 = \text{crc}_b(t)$. Based on the rolling property of the CRC function, we get:

$$\text{crc}_b(x \oplus z_1) = \text{crc}_b((t \ll b_x) | x) = \text{crc}_b(t \ll b_x) \oplus \text{crc}_b(x).$$

where b_x is the binary length of x , \oplus is XOR and $|$ is bitwise and. We also have

$$\text{crc}_b(y \oplus z_1) = \text{crc}_b(t \ll b_x) \oplus \text{crc}_b(y).$$

If $\text{crc}_b(x)$ and $\text{crc}_b(y)$ are both even (odd), $\text{crc}_b(t \ll b_x) \oplus \text{crc}_b(x)$ and $\text{crc}_b(t \ll b_x) \oplus \text{crc}_b(y)$ are both even or odd, that is, $\text{crc}_b(x \oplus z_1)$ and $\text{crc}_b(y \oplus z_1)$ are both even(odd).

It is the same for $\text{crc}_b(x \oplus z_2) = \text{crc}_b(y \oplus z_2)$

Proof of Theorem 2 (co-primes) Let U, W, U' and W' denote the distribution of the hashing output of $H^{m_1}, H^{m_2}, H^{q_1 m_1}$ and $H^{q_2 m_2}$, respectively. When $\hat{H} \gg q_1 q_2$ (condition 2 in Theorem 2), $\forall i' \in [0, q_1 - 1], j' \in [0, q_2 - 1], i \in [0, m_1 - 1]$, and $j \in [0, m_2 - 1]$, we have,

$$\Pr(U' = i', W' = j') = 1/(q_1 q_2)$$

When condition 1 is not satisfied in Theorem 2, there exists hash correlation brought by $q_1 m_1$ and $q_2 m_2$ because the mapping from $q_1 (q_2)$ to $m_1 (m_2)$ has a remainder. This mapping causes the non-uniform distribution of q_1 integers over m_1 slots, where $q_1 m_1$ slots get $\lfloor q_1/m_1 \rfloor + 1$ integers each, and $m_1 - q_1 m_1$ slots get $\lfloor q_1/m_1 \rfloor$ integers each. We denote this non-uniformity by the term *the approximation error*. Note that when q_1 increases, the difference between $(\lfloor q_1/m_1 + 1 \rfloor)/q_1$ and $(\lfloor q_1/m_1 \rfloor)/q_1$ can be reduced, and thus, we can quantify the approximation error by Equation (2).

$$\text{err} = (q_1 m_1)/q_1 + (q_2 m_2)/q_2 \quad (2)$$

where err denotes the approximation error. When condition 1 is satisfied, $\text{err} \approx 0$. Let $S(i) = \{i' | i' \% i = 0\}$, and $S(j) = \{j' | j' \% j = 0\}$, we have,

$$\begin{aligned} \Pr(W = j | U = i) &= \frac{\sum_{i' \in S(i), j' \in S(j)} P(U' = i', W' = j')}{\sum_{i' \in S(i)} P(U' = i')} \\ &= \frac{(q_1/m_1)(q_2/m_2)(1/(q_1 q_2))}{(q_1/m_1)(1/q_1)} \\ &= 1/m_2 = \Pr(W = j) \end{aligned} \quad (3)$$

Firebolt: Finding Bugs in Programmable Data Plane Generators

Jiamin Cao^{*}, Yu Zhou[†], Chen Sun[†], Lin He^{*}, Zhaowei Xi^{*}, Ying Liu^{*}

^{*}Tsinghua University [†]Alibaba Group

Abstract

Programmable data planes (DP) enable flexible customization of packet processing logic with domain-specific languages such as P4. To relieve developers from lengthy codes and tedious hardware details, many researches propose DP program generators that take high-level intents as input and automatically convert intents into DP programs. Generators must be correct, otherwise they may produce buggy programs or DP logic that is inconsistent with intents. Nevertheless, existing verification tools are designed to verify individual DP programs, not generators. They either cannot achieve high bug coverage or cannot debug generators with high scalability.

This paper presents *Firebolt*, a blackbox testing tool designed to dig out faults in DP program generators, including security vulnerabilities, intent violations, and generator crash. *Firebolt* achieves high bug coverage by using syntax-guided intent generation to construct a comprehensive, syntactically correct, and semantically valid intent set. To avoid intent explosion, *Firebolt* designs an intent space pruning approach that eliminates redundant intents while preserving representative ones. For high scalability, *Firebolt* automatically formalizes DP programs and intents for verification. We apply *Firebolt* to three popular open-source DP generators. Evaluation results demonstrate that *Firebolt* can detect $2\times$ bugs with 0.1% to 0.01% human efforts compared to existing tools.

1 Introduction

Programmable network devices [1, 2] together with domain-specific programming languages (e.g. P4 [3]) have enabled many in-data-plane (DP) network functions, such as monitoring [4–6], security [7–9], routing [10–12] and so on. Meanwhile, booming DP functions heavily burden programmers with lengthy codes (100s to 1000s lines of code, LoC [13]) and manual consideration of tedious hardware constraints [14]. To this end, a growing body of research proposes DP generators [14–27], which provide high-level declarative primitives to easily express *intents*, and a compiler to convert intents into platform-specific DP programs and table entries. DP generators can reduce LoC by 80% [14] with resource optimizations that would otherwise require manual efforts. Above benefits encourage researchers and industries to design various DP generators for network monitoring [16, 17] and even *mission-critical* functions like routing [14] or security [21, 22].

Considering the prevalence of DP generators, guaranteeing their *correctness* becomes a must-be-solved problem. However, our study (§2) reveals that three types of mistakes including program security vulnerabilities such as out-of-bound register access, intent-program inconsistency, and generator crash, may happen to advanced DP generators [16, 17, 21], which can result in serious mistakes such as missing attacks and undesired packet processing procedure.

Unfortunately, little attention has been devoted to guaranteeing the correctness of DP generators. Existing tools focus on finding security vulnerabilities in DP programs [28–34], or verifying the consistency between high-level intents and DP programs [28–31]. However, these tools are not designed for debugging DP generators and thus fall short in two aspects: (1) *Coverage*. Existing verification tools aim to verify individual intent-program pairs instead of finding all bugs in advanced DP generators. As intents are numerous, even infinite, verification tools can hardly cover all generator faults. (2) *Scalability*. To check intent-program consistency, verification tools require massive human-written specifications of intents (100s to 1000s of LoC) for one program, which is error-prone and time-consuming.

This paper presents *Firebolt*, a blackbox testing tool designed to dig out DP generator faults including security vulnerabilities, intent violations, and crash with high coverage and scalability. The key idea of *Firebolt* is thoroughly constructing intents as test cases to achieve high coverage, and automatically producing specifications of intents with little human intervention for verification to achieve high scalability.

However, realizing such a tool is challenging in three aspects: *intent generation* that should contain every reasonable intent, *intent explosion* that results in unacceptably long testing time due to numerous intents, and *intent diversity* that hampers automatic specification derivation for verification. In response, *Firebolt* proposes the following innovative designs.

- *Intent generation*. *Firebolt* should generate a comprehensive intent space containing every reasonable intent for high coverage. However, random composition of intent grammar symbols can produce infinite intents, which is impractical for testing. We first generate syntactically correct intents based on intent grammar in Backus-Naur form (§4.1). We then identify semantic dependencies between grammar symbols and filter semantically valid intents (§4.2).

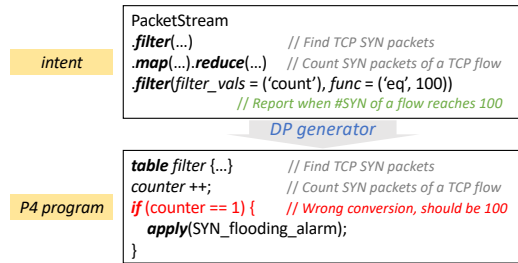


Figure 1: False SYN flooding alarms due to intent violation.

- *Intent explosion.* Due to wide parameter range and cyclic symbol reference, there may still exist massive or even infinite redundant intents that are syntactically correct and semantically valid, which compromises testing efficiency. To handle intent explosion, we design *intent space pruning* to eliminate redundant intents while keeping representative ones (§4.3). Remaining intents are input into generators to find crash bugs or to generate DP programs for verification.
- *Intent diversity.* The high diversity of intents and corresponding DP programs for one generator and across generators makes it challenging to devise a uniform approach for verification. To achieve high scalability, *Firebolt* first formalizes all DP programs into unified Z3 formulas [35] (§5.1). Next, instead of manually translating (1000s of) intents into specifications, we write specifications for intent grammar symbols, and automatically compose symbol specifications into the intent specifications (§5.2). Finally, we uniformly verify intent specifications and Z3 formulas to detect intent violations and security vulnerabilities (§5.3).

We apply *Firebolt* to three popular open-source DP generators, *i.e.*, Marple [16], Sonata [17], and Poise [21]. In all test cases of the three generators, *Firebolt* discovered 19 bugs including 3 security vulnerabilities, 13 intent violations, and 1 crash bug, while existing verification tools merely cover 10. Moreover, *Firebolt* requires 0.1% to 0.01% human-written LoC compared to existing tools under equal bug coverage.

2 Motivation

If a DP generator fails to faithfully translate programmer intents (intent violations), or produces logically flawed programs (security vulnerabilities), or even crashes under reasonable input intents, it adversely affects production efficiency and introduces instability into online DP functions. Below we present two example bugs that have been detected by *Firebolt* to reveal the consequences of faulty generators.

#1: False SYN flooding alarm due to intent violation. As presented in Figure 1, a SYN flooding monitoring function counts per-flow TCP SYN packet number. If a counter exceeds a threshold, a SYN flooding alarm is produced. However, due to a bug in Sonata [17], an advanced monitoring function generator, the threshold is wrongly configured as 1 instead of the originally intended 100, which results in massive false alarms that completely violates the monitoring goal.

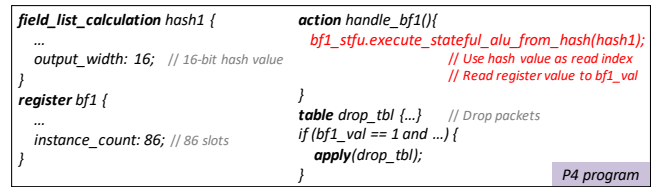


Figure 2: Connection legality misjudgement due to security vulnerabilities.

#2: Connection legality misjudgement due to security vulnerabilities. Poise [21] takes intents as input and generates context-aware security policies in programmable devices. It generates an in-DP bloom filter to track illegal connections. However, the bloom filter has 86 slots but is indexed by a 16-bit variable (0 to 65535), as shown in Figure 2. If the index exceeds 86, a random value outside the bloom filter will be returned. Such a vulnerability could incur wrong judgement of connection legality and lead to potential security leakage.

To eliminate above mistakes, programmers have to review generated DP programs and check intent-program consistency, which costs extra human efforts. Some recent tools are designed to find security vulnerabilities in DP programs [28–34] or verify the intent-program consistency [28–31]. However, using these tools to debug DP generators requires massive human efforts to verify each intent-program pair, and cannot fully cover all intents. Therefore, on modifying the intent or expressing a new intent, these tools must be repetitively executed to ensure program correctness. Unfortunately, doing so brings the scalability problem. To verify intent-program consistency, 100s to 1000s of LoC must be written manually to convert intents into specifications [16, 31]. Such LoC is comparable to the DP program, which is error-prone, time-consuming, and not scalable.

Instead of verifying individual programs, we propose *Firebolt* to debug generators. *Firebolt* thoroughly generates intents as test cases to detect generator faults, and automatically derives specifications from intents to improve scalability.

3 Overview

The key idea of *Firebolt* is thoroughly constructing intents as test cases to achieve high coverage, and automatically producing verification specifications to achieve high scalability. *Firebolt* workflow includes two major steps, *i.e.*, intent generation and program verification, as shown in Figure 3.

§4 - Intent generation. To thoroughly explore the intent space and cover all generator faults with little redundancy, *Firebolt* takes the intent grammar and semantic constraints of a DP generator as input and generates all possible and correct intents. Meanwhile, *Firebolt* losslessly prunes the generated intent space to eliminate redundancy and produces final test cases, which are then input into DP generators to generate DP programs or to find crash bugs.

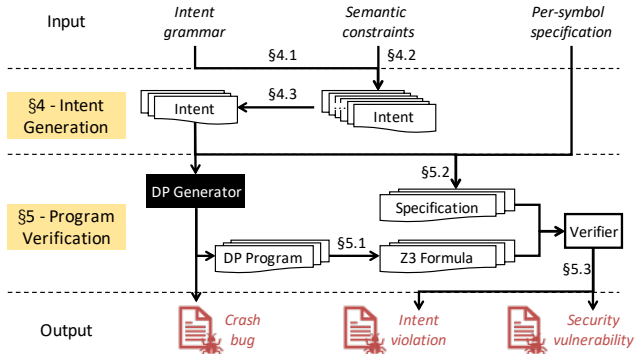


Figure 3: Workflow of Firebolt.

§5 - Program verification. To automatically verify the correctness of generated DP programs, the key is to derive the specification of intents for verification. *Firebolt* provides a general and high-level specification to express every intent grammar symbol, and automatically derives the specification of each intent based on per-symbol specifications. Meanwhile, *Firebolt* formalizes the output DP programs using Z3 formulas. Finally, *Firebolt* checks whether the Z3 formulas (1) are consistent with intent specifications and (2) have security vulnerabilities.

4 Intent Generation

The design goal of intent generation is producing a group of intents that (1) thoroughly covers all correct intents so that all generator faults can be discovered and (2) possesses little redundancy so that generator testing can be efficient. To achieve the above goals, we start by comprehensive intent generation with both syntactical correctness (§4.1) and semantic validity (§4.2) in mind. Next we analyze the source of redundant intents and propose the intent space pruning approach (§4.3).

4.1 Syntax-Guided Intent Generation

A DP generator must expose the intent grammar for expressing intents. For example, as shown in Figure 1, Sonata [17] provides primitives like *filter*, *map*, and *reduce* and parameters like *eq* and *count*. The intent grammar describes lawful function calls, parameter ranges, and syntax, which serves as the foundation to generate possible intents from scratch.

We refer to syntax-guided synthesis [36, 37], a common approach in program synthesis that finds the desired program by searching the program space described by a grammar. We leverage its idea and redefine the intent generation problem as: given the grammar G of a DP generator, we need to explore the intent space and generate all syntactically correct intents.

Intent grammar formalization. Syntax-guided intent generation takes grammar as input. However, different generators can provide grammar in various formats [14, 16, 17]. We need to formalize grammars of generators into a unified expression. We observe that Backus-Naur form (BNF) [38] is the most common context-free grammar (CFG) [39] for describing the

Algorithm 1 Syntax-Guided Intent Generation

```

1: function SYGUG( $G$ )
2:    $Q = \text{Queue}(G.n_r)$  ▷ Initialize
3:   while  $Q.size() > 0$  do
4:      $n = Q.front()$ 
5:      $Q.pop()$ 
6:     if  $n.has\_nt()$  then
7:        $A = n.first\_nt$ 
8:       for  $A \rightarrow \beta \in G.R[A]$  do
9:          $n = \alpha A \gamma \xrightarrow{A \rightarrow \beta} n_1 = \alpha \beta \gamma$  ▷ Grow graph
10:         $Q.push(n_1)$ 
11:      end for
12:    else
13:      OUTPUT( $n$ ) ▷ Output generated intent
14:    end if
15:  end while
16: end function

```

syntactic structure of programming languages. Most DP generators [14, 16, 21–23] provide a BNF syntax specification. Thus, *Firebolt* uses BNF for grammar formalization.

Note that *Firebolt* can also work with non-BNF grammars by adopting the expansion rules of these grammars during syntax-guided intent generation.

Preliminaries of BNF. Dark rectangle in Figure 4 shows partial BNF expression of Marple [16] intent grammar. A grammar G expressed in the BNF format is a quadruple $\langle N, S, \Sigma, R \rangle$, where (1) N is a finite set of non-terminal symbols that can be expanded to one or more terminal and non-terminal symbols, (2) S is the start symbol in N , (3) Σ is a finite set of terminal symbols that can appear in an intent, (4) R is a finite subset of $N \times (N \cup \Sigma)^*$, where each member $(A, \beta) \in R$ is called an expansion rule and is written as $A \rightarrow \beta$. A sequence of non-terminal and terminal symbols in $(N \cup \Sigma)^*$ is called a *sentential form*, which represents an intermediate intent.

Syntax-guided intent generation. With a grammar G in BNF format, a possible intent must start with S and then be replaced with expansion rules in R until there are no non-terminal symbols that need to be expanded, *i.e.*, it ends with a composition of terminal symbols in Σ . Therefore, generating all possible syntactically correct intents can be visioned as growing a single-rooted (S) graph with one or more rules in R , and collecting all leaf nodes ended with any one symbol in Σ .

We formalize the above process as follows. We define the intent generation graph $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ as a directed labeled graph derived from grammar $G = \langle N, S, \Sigma, R \rangle$, with the nodes $\mathcal{N} \subseteq (N \cup \Sigma)^*$ and the edges $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N} \times R$. Each node in \mathcal{N} has a sentential form which can be derived from the start symbol S . Each edge in \mathcal{E} represents a non-terminal symbol expansion according to an expansion rule. At node n_1 whose sentential form is $\alpha A \gamma$, where A is a non-terminal symbol, we can apply the expansion rule $A \rightarrow \beta \in R$ and derive a child node n_2 with a new sentential form $\alpha \beta \gamma$. \mathcal{G} has a root node n_r with sentential form S , and many (maybe infinite) leaf nodes.

Algorithm 1 depicts the procedure of growing graph \mathcal{G} using *depth first search*. We maintain a *queue* Q to store nodes in \mathcal{G} . First, Q is initialized with the root node n_r (line 2). Then, in each iteration (line 3-15), we pop the first node n in Q . If n has no non-terminal symbols in its sentential form, we output n as an intent (line 13). Otherwise, we find the first non-terminal symbol A in n , and apply all possible expansion rules of A to generate child nodes (line 9), which are then appended to Q for further expansion. We repeat this process until Q is empty and all syntactically correct intents are generated.

4.2 Semantic Constraint Injection

Syntax-guided intent generation can cover syntactically correct intents. Nonetheless, some syntactically correct intents do not make sense, or, say, are semantically invalid [40]. Below we first introduce two types of semantically invalid intents. Next we identify semantic constraints between grammar symbols, and present the semantic constraint expression and injection mechanisms to filter semantically valid intents.

Semantically invalid intents. Besides conforming to the syntax, intents must also comply with semantic constraints. Below we identify two types of semantically invalid intents.

- *Uncompilable intents.* Syntactically correct intents cannot guarantee successful compilation. Typical examples include a variable that is not declared before it is referenced, a variable reference whose dimension is inconsistent with the declaration, or a variable that is repeatedly defined. Numerous such intents violate the semantic constraints of the intent grammar, and therefore should be ruled out.
- *Incomplete intents.* Some intents are semantically incomplete. For example, the *map* primitive in Marple [16] distributes incoming data according to certain match fields, and assigns a computing expression to process a temporary variable, which is meaningless if it is not referenced later. Therefore, an intent with a *map* primitive in the end of a query is considered incomplete and should be ruled out.

Semantic constraint identification. The existence of semantically invalid intents indicates that the above mentioned syntax-guided intent generation graph contains unreasonable leaf nodes (incomplete intents) or even invalid expansion rules (branches) for intermediate nodes (uncompliant intents). Thus, we should identify semantic constraints of intent grammar, and leverage the constraints to supervise the intent generation process. Overall, we classify the constraints into *exclusion constraints* and *dependency constraints*.

- *Exclusion constraints:* indicate that if an expansion rule r_1 on a node n_1 exists, the expansion rule r_2 on a node n_2 is not valid. We formally express them as:

$$\text{if } \exists r_1 \text{ on } n_1, \text{ then } \nexists r_2 \text{ on } n_2$$

A typical example is that one variable name (such as the name of a packet stream) cannot be defined repeatedly.

- *Dependency constraints:* indicate that only if an expansion rule r_1 on a node n_1 exists, the expansion rule r_2 on a node

```

(prog)          ::= (aggFun)* (streamStmt)+           Marple.bnf
(streamStmt)    ::= (streamName) = (streamQuery)
(streamName)    ::= R(number)
(streamQuery)   ::= (map) | ...
(map)          ::= map ((streamName), [(mapCol)], [(mapExpr)])

```

Example 1: StreamName cannot be defined repeatedly

```

if ∃ (r1) on (n1), ∄ (r2) on (n2), (n2) ↔ (n1)
(n1) : *, (prog)(streamStmt)(streamName),*
(r1) : (streamName) →*
(n2) : *, (prog)(streamStmt)(streamName),*
(r2) : (r1)

```

Example 2: Map query operates on a stream that has been defined

```

if ∃ (r1) on (n1), ∃ (r2) on (n2), (n2) → (n1)
(n1) : *, (prog)(streamStmt)(streamQuery)(map)(streamName),*
(r1) : (streamName) ↦ T
(n2) : *, (prog)(streamStmt)(streamName),*
(r2) : (r1)
(n1).(streamStmt) ≠ (n2).(streamStmt)

```

Figure 4: Semantic constraint examples in Marple [16].

n_2 is valid. We formally express them as:

$$\text{if } \exists r_1 \text{ on } n_1, \text{ then } \exists r_2 \text{ on } n_2$$

A typical example is that a *Map* primitive must operate on a stream that has been previously defined.

With the above classification in mind, we thoroughly analyze the intent grammar and derive semantic constraints. Missing constraints can result in some semantically invalid intents left as test cases and slightly compromise the testing efficiency, which is considered acceptable. However, wrongly-written constraints will incur wrong deletion of semantically valid intents and impair generator fault coverage. Therefore, we must guarantee the correctness of the constraints. We have investigated several advanced generators [16, 17, 21] and observe that each of them corresponds to <20 semantic constraints, which is acceptable for manual inspection.

Semantic constraint expression and injection. Identified semantic constraints should be uniformly encoded for injection. To clearly express a constraint, we need to clearly specify the constraint type (\exists or \nexists) and elements (r_1 , n_1 , r_2 , and n_2). Next we showcase two semantic constraints in Marple [16].

Example 1 shown in Figure 4 presents an exclusion constraint. A Marple program $\langle prog \rangle$ includes multiple streams $\langle streamStmt \rangle$, each with name $\langle streamName \rangle$. The names of streams should not be defined repeatedly. n_1 and n_2 describe the node where stream names are defined. Naturally, there exists an expansion trace from the start symbol $\langle prog \rangle$ to the current symbol $\langle streamName \rangle$. We use \leftrightarrow to indicate that the two nodes can appear in any order on the path. We use $*$ to match any expansion traces in n and any expansion rules in r . By making $r_2 = r_1$, the constraint prevents $\langle streamName \rangle$ from taking duplicate values of r_1 in any other node n_2 .

Example 2 in Figure 4 presents a dependency constraint. If a $\langle map \rangle$ query in Marple operates on a stream with a name other than T (the name of the original input stream in Marple),

this stream should be previously defined. We use $n_2 \rightarrow n_1$ to constrain the ordering between two nodes, *i.e.*, n_2 should be an ancestor node of n_1 . By making $r_2 = r_1$, the constraint guarantees that a stream is defined before referenced.

We present all identified semantic constraints of Marple [16], Sonata [17], and Poise [21] in Appendix A. During intent generation, an expansion rule r on node n is rejected if it violates exclusion constraints, and a leaf node is rejected if not all dependency constraints are satisfied on its path.

4.3 Intent Space Pruning

Despite we guarantee the syntactical correctness and semantic validity of intents, the massive expansion rules and their combinations may still build an extremely large or even infinite intent space, due to two reasons, as illustrated in Figure 5.

- *Wide parameter range.* A non-terminal symbol may have many possible expansion rules, which corresponds to a large node degree. For example, a 16-bit integer has 65536 possible values. Worse still, if a sentential form has multiple such non-terminal symbols, the exponential combination can lead to an explosion of the intent space.
- *Cyclic symbol reference.* A non-terminal symbol may return to itself after expansion, *i.e.*, the cyclic symbol reference, which corresponds to an infinite depth of the intent derivation graph. For example, an arithmetic expression has an expansion rule of $\langle S \rangle ::= \langle S \rangle + \langle S \rangle$. The circular expansion leads to an infinite number of possible intents.

For all generated intents of a DP generator, we observe that most intents would not cause any bugs, while many could cause the same bug. To strike a balance between coverage and efficiency, we propose the following two mechanisms to prune the intent space without losing intent representativeness.

Method #1: Intra-symbol representativeness. To handle wide parameter range, we propose to *keep representative expansion rules*, which include three categories.

- *Boundary rules.* Boundary values in numbers, including minimum and maximum values (e.g., 0 and 65535 for a 16-bit parameter), usually represent some extreme cases or conditions, and should be included in the test cases.
- *Random rules.* In addition to boundary values, we should take random values from values other than boundary values (e.g. one value from 1 to 65534 for a 16-bit parameter).
- *Previously selected rules.* When the same non-terminal symbol is expanded multiple times in a sentential form, the choices of expansion rules are actually correlated. In this case, the previously selected random rules should also be included in the latter non-terminal symbol expansions. For example, Marple [16] uses the query name to identify a query. Suppose a former expansion rule defines the name of a query Q , and a latter expansion rule references the name of a query (maybe query Q , maybe not), value Q should be included in the latter rule to keep representativeness.

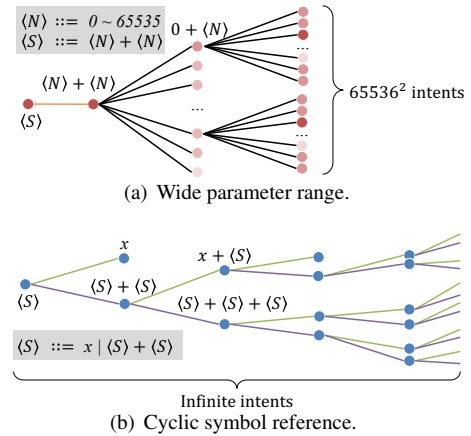


Figure 5: Two types of intent space explosion.

Method #2: Inter-symbol combination representativeness.

For cyclic symbol reference, we can think of an expansion circle as a non-terminal symbol returning to itself with zero to many intermediate non-terminal symbols. To handle cyclic symbol reference induced intent explosion, we should break infinite symbol recurrence without losing representativeness.

To this end, we refer to the combinatorial testing (CT) theory [41] for software testing in the software engineering field. Provided that a software is composed of multiple features, the CT theory indicates that a minimal set of test cases for the software should include *individual* features and combinations of *two* distinct features, which are enough to effectively test the software and find most bugs. If a test case containing three or more features causes a software bug, the root cause may still lie in the interaction of two features among them.

Inspired by the CT theory, to effectively test the DP generator with high efficiency, we can first extract all distinct features, *i.e.*, combination of non-terminal symbols, according to the intent grammar. Then, we prune the intent space and only keep sentential forms that are either (1) individual features, or (2) possible combinations of n distinct features, where $n = 2$. Our evaluation results in §6.2 reveal that using a higher combination factor (e.g. $n = 3$) cannot find more bugs in the DP generator, which proves the effectiveness of applying the CT theory for DP generator testing.

Finally, we introduce how we extract distinct features, *i.e.*, combinations of non-terminal symbols. Recall that the CT theory limits the recurrence of the same feature to two times. Therefore, each feature should only include distinct symbols. Suppose an intent grammar has k non-terminal symbols. By picking a random number (1 to k) of distinct symbols and organizing them in all possible sequences, we can generate $N = \sum_{i=1}^k A_i^i$ features where A stands for the permutation symbol, *i.e.*, $A_n^m = n!/(n-m)!$. With the combination factor $n = 2$, there exist at most $(N + A_N^2)$ sentential forms composed of non-terminal symbols. Suppose there are s terminal symbols and p parameter value options, we produce at most $(N + A_N^2) \times s \times p$ test cases, which are *finite* and feasible for testing. Our evaluation results in §6.2 show that using the

above pruning methods, *Firebolt* will generate <10K intents for testing three advanced DP generators [16, 17, 21].

So far, we have thoroughly explored the intent space to generate syntactically correct and semantically valid intents with little redundancy. We feed these reasonable intents into the DP generator to find crash bugs or to generate DP programs for verification, which we will introduce in the next section.

5 Program Verification

In this section, we introduce how *Firebolt* verifies the correctness of the generated DP programs from two aspects, *i.e.*, whether there are potential security vulnerabilities, and whether the DP programs are consistent with corresponding intents. We use Z3 [35], a Satisfiability Modulo Theories (SMT) solver, which can take (1) Z3 *formulas* and (2) Z3 *assertions* as input, and formally verify whether the formulas satisfy the assertions. In the rest of this section, we first introduce how we automatically formalize the generated DP programs as Z3 formulas (§5.1). Next, to avoid manually converting 1000s of intents into Z3 assertions, we provide a general and flexible specification to express every symbol of the intent grammar, and automatically compose symbol specifications into intent specifications, which will then be converted into Z3 assertions (§5.2). Finally, we check intent-program consistency and detect security vulnerabilities (§5.3).

5.1 DP Program Formalization

We use the popular P4₁₆ language as an example to illustrate how to formalize DP programs into Z3 formulas. P4₁₄ programs can be first converted into P4₁₆ programs using open-source P4 compiler suite [13] and then formalized into Z3 formulas by *Firebolt*. Gauntlet [42] has proposed approaches to convert partial P4 programs into Z3 formulas, but does not cover the formalization of P4 *table entries* and *externs*, which are essential to faithfully convert P4 programs. Our formalization solution is built atop Gauntlet. Below we first introduce the idea and capability of Gauntlet, and then introduce how we formalize externs and table entries.

Formalizing each programmable block. A P4₁₆ program is composed of several programmable blocks (*e.g.*, packet parser, ingress control flow, egress control flow, packet deparser, etc.). We provide an example of a P4 *match-action table* residing in the control flow block in the head of Figure 6. For each block, Gauntlet performs the following conversion.

- *Input parameter* → *free Z3 variable*. Two special types of Z3 variables, *i.e.*, Z3_INVALID and Z3_UNDEFINED, are defined to represent invalid and undefined parameter values.
- *Function* → *Z3 operation* that refers to input Z3 variables. For example, a table lookup function is converted into a reference to the resulting action index. Operations like parameter initialization or invalidation can refer to Z3_INVALID and Z3_UNDEFINED special variables.

Part of P4₁₆ Program: Match-Action Table

```
action a0 (z) {y = z;}
table t0 {
  key = k0 : exact;
  actions = {
    no_op;
    a0;
  }
  default_action = no_op;
}
```

Table entries of t0:	
1 =>	a0(1)
2 =>	a0(2)

Input free Z3 variables:

```
(_ BitVec 32) k0 // Match key of table t0
(_ BitVec 32) t0_index // Action index of table t0
(_ BitVec 32) a0_z // Parameter of action a0
```

Output Z3 expressions (with table entries):

```
(_ BitVec 32) y = (ite (= k0 1) 1 (ite (= k0 2) 2
Z3_UNDEFINED))
```

Output Z3 expressions (without table entries):

```
(_ BitVec 32) y = (ite (= t0_index 1) a0_z Z3_UNDEFINED)
```

Figure 6: Examples of formalizing match-action tables.

- *Output parameter* → *output Z3 expression* that is the result of executing Z3 operations on the input Z3 variables.

Formalizing table entries. Gauntlet assumes that the contents of the table are unknown, and does not include the configuration of table entries in the output Z3 expressions. However, generating correct table entries is also critical for the DP generator, as table entries also reflect intents. For instance, a *filter*(ip.src=192.168.1.1) intent segment in Sonata indicates that subsequent operations will operate on special flows, which corresponds to a table entry in generated DP programs.

To formalize table entries, we use a nested if-then-else statement to imitate a match-action table call, as shown in Figure 6. When table entries are provided, for each parameter modified by the table (parameter *y* in this example), we use an if-then branch to express the modification in Z3 expressions as follows: if the key of incoming data matches a specific table entry ($k0 = 1$ or $k0 = 2$), the corresponding action is executed ($y = 1$ or $y = 2$). If the key does not match any entry, the default action is executed (y is not assigned an initial value, and is therefore undefined). When no table entry is specified, we assume that all actions in the table are executable. We use a free Z3 variable (*t0_index*) to indicate the index of the action to be executed, and a separate free Z3 variable for each action parameter (*a0_z* for parameter *z*). A table call can then be represented by a nested if-then-else statement with each branch representing the execution of one action.

Formalizing externs. P4 programs often operate on extern objects such as stateful memory and hash calculations. Gauntlet interprets externs as a function call that returns an arbitrary value. However, an accurate translation of externs is critical, since externs can maintain program internal states and may be modified and referenced. For example, the SYN flooding alarm program shown in Figure 1 maintains a counter that will later be compared to a threshold, which should be embedded in the Z3 formula. Below we introduce our approaches to handle stateful memory and hash calculation, respectively.

```

Part of P416 Program: Register Reading and Writing
register<bit<32>>(32w1024) reg;
reg.read(r_index, r_value);      // r_value = reg[r_index]
reg.write(w_index, w_value);    // reg[w_index] = w_value

Input free Z3 variables:
(_ BitVec 32) reg_read_value
(_ BitVec 32) r_index, w_index, w_value
Output Z3 expressions:
(_ BitVec 32) reg_instance_count = 32w1024
(_ BitVec 10) reg_write_index = w_index
(_ BitVec 32) reg_write_value = w_value
(_ BitVec 10) reg_read_index = r_index
(_ BitVec 32) r_value = reg_read_value
(a) Register reads and writes

Part of P416 Program: Hash Calculation
// Update hash_table_index with hash value
hash(hash_table_index, HashAlg.crc32, 32w0, inKey, 32w1024);

Input free Z3 variables:
(_ BitVec 32) crc32_hash_value // Hash value
(_ BitVec 96) inKey           // Hash key
Output Z3 expressions:
(_ BitVec 32) hash_table_index = crc32_hash_value
(_ BitVec 32) crc32_hash_width = 32w0
(_ BitVec 128) crc32_hash_field = inKey
(_ BitVec 32) crc32_hash_size = 32w1024
(b) Hash calculation

```

Figure 7: Examples of converting externs into Z3 formulas.

- *Stateful memory.* We take register, an indexed array of stateful cells, as an example to illustrate our approach, which also applies to other types of stateful memory such as counters. A naive method to formalize a register is to define a free Z3 variable to represent the initial value and generate an output Z3 expression to represent the new value for each cell in the register. In this way, register reading can be converted into referencing the Z3 variable, and register writing can be converted into updating the Z3 expression. Then formalizing a register array with n instances requires $2 * n$ Z3 variables and expressions. Furthermore, we notice that for most commercial switches, a register array can be read/written only once in the switch pipeline. Thus, we only need to maintain the index and values for at most two register cells, as shown in Figure 7(a). Besides, we use another Z3 variable to store the size of the register, which can be used to detect out-of-bound register access in §5.3.
- *Hash calculation.* Hash is used to map large data to fixed-size values. We define a free Z3 variable to represent the computed hash value. Meanwhile, we store the parameters that impact the hash value, e.g., the hash key and hash size, in the output expressions, as shown in Figure 7(b). Then, we can flexibly adjust the effect of the hash mapping by imposing the mapping relationship between the hash value and hash parameters when checking the Z3 expressions. For example, a conflict-free hash implies the one-to-one mapping between the hash key and hash value. In this way, we can avoid the complex hardware-specific hash calculation while maintaining the properties of hash operations.

5.2 Intent Formalization

The intent generation process can produce thousands of (§6) intents for one DP generator. Manually converting intents that are composed of different symbols of the same generator, or intents belonging to different generators, is time-consuming and not scalable. We observe that intents are generated by expanding non-terminal symbols. Therefore, instead of converting each intent, our key idea is first writing the specifications of each symbol in the grammar, and then automatically composing symbol specifications into intent specifications, which will finally be converted into Z3 assertions.

Symbol specification. To uniformly express highly-diversified intent grammar symbols across generators, we need to design an expression format that should be *general* enough to specify various symbols, and *flexible* enough for composition. The reason why we do not directly use Z3 assertions as the specification is that Z3 assertions are logical expressions that are low-level and counter-intuitive.

We propose to uniformly express each symbol as a high-level *function* written in python-like expressions. The function specification satisfies above requirements. It is general enough to specify the format and semantics of input, logic, and return values of individual symbols, and flexible enough for composition by sequentially performing function calls and correlating input and output of different functions.

Specifically, we regulate that one function consists of two segments, a declaration function `DECL_FUNC` that defines internal states of symbols, and an execution function `EXEC_FUNC` that describes the processing logic of input parameters, internal states, and output values. Stateless symbols such as a flow *filter* maintain no internal states and therefore can be expressed with only `EXEC_FUNC`. A typical execution function often takes network packets as input, and starts by `PARSE`-ing input packets into a series of header fields, e.g., Ethernet \rightarrow IPv4 \rightarrow TCP. For each packet, we also generate standard metadata fields regulated by the P4 grammar, such as the output port. Next, symbol logic can operate on packet headers, metadata fields, and symbol internal states by packet modification, counting, forwarding, or other actions, and finally provide return values or return directly.

Figure 8 takes the `<groupby>` symbol in Marple [16] as an example to illustrate how to construct a stateful symbol specification. An entire `<groupby>` symbol is formatted as `<groupby> ::= groupby(<stream>, <columns>, <aggFunc>)`, which groups *streams* according to specific *columns* with the aggregation function *aggFunc*. In the declaration function `DECL_FUNC`, a key-value storage is declared for each possible option `<var>` in the child symbol `<aggFunc>`. The `EXEC_FUNC` updates the internal states with no packet parsing, modification, or forwarding. First, we initialize a variable *tuple* with the tuple contained in the input stream whose name is `<streamName>`. Then we get the aggregation key from the aggregation field in `<columns>`. Using the aggregation key,

```

# (groupby) ::= groupby ((streamName), (columns), (aggFunc))
# (aggFunc) ::= def (aggFun) ((vars), (columns)): (codeBlock)
DECL_FUNC() =
  states = []
  for var in <aggFunc>.<vars>.exec():
    KEY_VALUE_STORAGE REG_NAME_var
    states.append(REG_NAME_var)
EXEC_FUNC(stream_list) =
  tuple = <streamName>.exec(stream_list)
  key = tuple[<columns>.exec()]
  old_state = [reg[key] for reg in states]
  new_state = <aggFunc>.exec(old_state, tuple)
  states.update(key, new_state)
  tuple.append(new_state)
  return tuple

```

Figure 8: An example for constructing stateful specifications of non-terminal symbols: $\langle groupby \rangle$ in Marple [16].

we read the old states and execute the aggregation function $\langle aggFunc \rangle$. The output values of the aggregation function are used to update states of $\langle groupby \rangle$. Also, the output states are included in *tuple* for future usage in subsequent symbols.

Symbol specification composition. For each generated intent, *Firebolt* constructs its specification based on its generation path, i.e., the non-terminal symbol expansion process and the final terminal symbols. Starting from the semantics of the start symbol, *Firebolt* recursively appends the semantics of child symbols in the expansion rules by sequentially connecting their **FUNCs** to automatically construct the final specification.

Specification conversion into Z3 assertions. A Z3 assertion is a series of algebra expressions connected with logical operators, e.g. $(x > 10) \&\&(y < 5)$. We use a special type of Z3 assertions, i.e., $implies(f, g)$, to verify DP programs. $implies$ is an implication operator, which assumes a condition expression f on the input Z3 variables and asserts that the output satisfies the implication expression g . After expressing intents as a series of functions, we can identify how each parameter (e.g., header fields, forwarding port, and internal states) is modified by the functions. Therefore, we develop a tool that can automatically convert the functions into the $implies$ Z3 assertions that will be used to verify the DP programs.

5.3 Program Correctness Verification

With the Z3 formulas representing semantics of DP programs and intents formalized into Z3 assertions, we first check whether each P4 program is consistent with the corresponding intent. Then, we summarize security vulnerabilities from the literature, and introduce how to detect them in each program.

Intent-program consistency. To verify consistency, we take the Z3 formulas as input and use the Z3 constraint solver [35] to verify the Z3 assertions, i.e., a set of $implies(f, g)$. Specifically, we need to check the following three types of consistency, each with a different set of f and g .

- *Packet parsing consistency.* It indicates that parsed headers and header fields are ordered consistently with the original

intent, and each header field is parsed correctly. During DP program formalization, when formalizing the parser block, *Firebolt* treats the entire input packet header as a free Z3 variable, and outputs individual Z3 expressions to represent different header fields. We verify the parsing consistency by asserting that for each header field in the parsing part of the intent specification, (1) there is a corresponding output Z3 expression, (2) a failed parsing implies an invalid value. For example, the first 16 bits of the Ethernet header should be 0x800. The parsing of Ethernet fails if an Ethernet header does not start with 0x800, and (3) a successful parsing implies a correct parsed value. For example, given the condition that the first 16 bits of the header are 0x800, we should be able to obtain the Ethernet.dstAddr field by extracting specific bits from the input header variable.

- *Packet deparsing consistency.* It refers to the correct order and values of headers and header fields in the output packet. The checking of the deparser block is similar to the parser block. We omit details here for brevity.
- *Packet processing consistency.* Packet processing includes packet modification, forwarding and state updates, and corresponds to the behavior of several programmable blocks of a P4 program, e.g., both the ingress control flow and the egress control flow. To construct a complete Z3 formula, we first concatenate the Z3 formulas of individual related programmable blocks into a complete block. For each output Z3 expression of a block, if it is an input Z3 free variable of latter blocks, we replace the corresponding input with the current output, and recompute the output expressions of latter blocks. We iterate this process until the output of a block is no longer referenced by any blocks, which becomes the final output of the DP program and completes the Z3 formula for packet processing. Then, for each modified packet field and metadata in the Z3 formula, we extract related operators and construct an individual Z3 expression, which can be checked against corresponding intent specifications.

Security vulnerabilities. Security vulnerabilities are intrinsic flaws of DP programs without corresponding intents. For example, out-of-bound register access may cause unexpected behaviors and even online risks, and is never intended. We observe that security vulnerabilities can be converted into special Z3 assertions and verified against DP programs as introduced above. Therefore, we summarize security vulnerabilities highlighted by previous literature [28–32, 34], and introduce how to express them as Z3 assertions.

- *Invalid header access* may occur when the validity of a header is not checked before referencing it. To detect this bug, for each output Z3 expression, we assert that (1) each referenced header (Z3_h) belongs to a branch in an if-then-else statement, and (2) the if-condition in this branch includes a validity check (i.e., $Z3_h \neq Z3_INVALID$).
- *Implicit packet drops* occur when the egress_port is not specified. To detect this bug, in the output Z3 expression

Table 1: (1) Bugs detected by *Firebolt*, and (2) efficiency of *Firebolt* when debugging the three DP generators.

DP Generator Under Test	# Generated Intent	# Detected Bugs / # Intents Causing Bugs			Human-written LoC			Test-Case Size (Min / Max)			Running Time (Total / Average)	
		Crash Bug	Security Vulnerability	Intent Violation	Intent Grammar	Semantic Constraints	Per-Symbol Specification	Intent LoC	P4 Program LoC	# Table Entries	Intent Generation	Program Verification
Marple	7341	1 / 12	1 / 7329	2 / 23	93	70	323	1 / 32	211 / 481	0 / 0	168s / 23ms	1204s / 164ms
Sonata	7912	0 / 0	2 / 7912	5 / 243	34	10	178	1 / 19	253 / 375	7 / 43	27s / 3ms	926s / 162ms
Poise	2362	0 / 0	2 / 2362	6 / 362	25	25	132	1 / 12	704 / 893	1 / 12	23s / 10ms	355s / 150ms

of `egress_port`, we assert that no branch results in the undefined special variable, *i.e.*, `Z3_UNDEFINED`.

- *Out-of-bound register access* occurs when the read/write index exceeds the register array size. Since we use separate output Z3 expressions to record the read/write index and the array size in §5.1, we can detect this bug by comparing the array size and the index range. For direct access, where *index* is assigned an exact value, we assert that $index < size$. For non-direct access, *i.e.*, the *index* expression includes symbolic variables, such as hash values, we assert that the range of the symbolic variables is within the allowed range.
- *Decapsulation errors* happen when invalid headers are deparsed in the deparser. To detect this bug, we assert that for each output header expression, no branch results in the invalid special variable (*i.e.*, `Z3_INVALID`).
- *Forbidden writes* happen when a P4 program tries to write certain metadata values which are read-only, but the P4 compiler allows the program to write them. To detect this bug, we assert that for all read-only metadata fields, the values remain unchanged, *i.e.*, the output values are the original undefined values (*i.e.*, `Z3_UNDEFINED`).

6 Evaluation

We implement *Firebolt* with ~1200 lines of Python code for intent generation, and ~800 lines of C++ code for program verification. Our verifier is built atop Z3 [35], and can verify both P4₁₆ and P4₁₄ programs with the aid of P4 compiler suite [13]. All experiments were conducted in a Ubuntu 16.04 virtual machine with 4GB RAM and two 2.3GHz CPU cores.

We use *Firebolt* to test three popular open-source DP generators, including two for network telemetry, *i.e.*, Marple [16, 43] and Sonata [17, 44], and one for security policy enforcement, *i.e.*, Poise [21, 45]. Marple and Sonata both use sequential composition of data flow operators to construct telemetry queries, while Marple is more complicated by supporting self-defined variable names in each query and dependencies between queries. Poise is relatively simpler and enforces security policies by filtering customized packet header fields. Finally, we implement two advanced DP program verification tools (*i.e.*, Aquila [31] and p4v [30]) for comparison.

Our evaluation intends to answer the following questions.

- *Bug coverage.* We first discuss all discovered generator bugs by *Firebolt*. (§6.1) Next, we prove the bug coverage of *Firebolt* by (1) showing the intent representativeness of *Firebolt*, and (2) comparing the number of bugs discovered

by *Firebolt* and existing verification tools over open-source intents and programs of the generators. (§6.2)

- *Efficiency.* We introduce (1) the human efforts required by *Firebolt* to debug the generators, (2) the size of intents, P4 programs, and table entries that are generated and verified by *Firebolt*, and (3) the running time of intent generation and program verification. (§6.3)
- *Scalability.* We first compare the human efforts, *i.e.*, lines of hand-written codes, required by *Firebolt* and existing verification tools to debug the three generators. Then we evaluate the time required by *Firebolt* when verifying larger programs and more table entries. (§6.4)

6.1 Bug Analysis

As shown in Table 1, we find that all three generators have bugs, and discover 5 security vulnerabilities, 13 intent violations, and 1 crash bug in total. Below we introduce the detected bugs. To the best of knowledge, this is the first effort that comprehensively analyze and reason DP generator faults.

Security vulnerability. *Firebolt* finds security vulnerabilities in all generated programs of all three DP generators.

- *Invalid header access* is a common bug. All generated programs refer to some headers without checking validity.
- *Out-of-bound register access* is found in Poise, which may use a hash value that exceeds the size of the the register as the read/write index for the register.
- *Implicit drops* happen in Sonata. Generated programs never explicitly specify the egress port of input packets.

Intent violation. Due to the high intent diversity, intent violations are the most insidious bugs that cannot be easily detected by the developers of DP generators. Next we introduce intent violations in the three DP generators, respectively.

For Sonata, *Firebolt* finds 5 types of intent violation bugs in 243 generated programs out of 7912 programs in total.

- *Bug #1: Incorrect query combination.* When a *filter* query with the *eq* (=) function follows a *reduce* query, Sonata converts it into comparing the result of *reduce* and the default value of 1, regardless of the true value in the filter query. This bug may lead to false attack alarm (§2).
- *Bug #2, #3, #4: Missing/Incomplete table entries.* Sonata designs a *filter(k, v, f)* symbol to filter packets whose key (*k*) fields satisfy function (*f*) of value (*v*) fields. To implement a *filter* query in DP programs, both match-action tables and table entries should be generated. However, there are three

Table 2: The number of detected bugs and generated intents with different intent space pruning methods.

Pruning Method	# Detected Bugs			# Generated Intents			
	Marple	Sonata	Poise	Marple	Sonata	Poise	
Intra-Symbol	None	① ① ①② ②	①	> 50K	> 50K	> 50K	
	$r=1$	①①①② ①②①②③④⑤	①②①②③④⑤⑥	7341	7912	2362	
	$r=2$	①①①② ①②①②③④⑤	①②①②③④⑤⑥	14812	12384	3804	
Inter-Symbol	None	①	①② ②③	①②①②③④⑤⑥	> 50K	> 50K	2362
	$n=1$	①①①② ①② ②③④⑤	①②①②③④⑤⑥	2346	3523	2362	
	$n=2$	①①①② ①②①②③④⑤	①②①②③④⑤⑥	7341	7912	2362	
	$n=3$	①①①② ①②①②③④⑤	①②①②③④⑤⑥	> 50K	> 50K	2362	

○ Crash Bug ● Security Vulnerability ● Intent Violation

cases where table entries can be missing or incomplete. First, when a *filter* query operates on a variable, e.g., a counter, table entries are forgotten. Second, when a *filter* query has a *mask* function, Sonata translates it into an LPM table, but forgets to include the prefix length in table entries. Third, when a *filter* query has a *geq* (\geq) function and does not follow a *reduce* query, no table entries are generated.

- *Bug #5: Incorrect mask translation.* Sonata uses bit-wise AND for *mask* operations in a *map* query. It translates the mask m into $0xFF...F$ (F occurs $m/4$ times). When the mask length is not a multiple of 4, the translation is incorrect.

For Poise, *Firebolt* finds 6 types of intent violation bugs in 362 generated programs out of 2362 programs in total.

- *Bug #1: Incorrect list comparer.* Poise provides list comparer (*in* and *notin*) to check whether a value is in a list. However, *notin* is wrongly equated with *in* when translated.
- *Bug #2, #3, #4: Incorrect comparison operator.* Poise translates the comparison operators ($>$ and $<$) into a match-action table with *range* match and a table entry representing the comparison range. However, this range incorrectly includes the boundary value that should be excluded. That is, $>$ and $<$ are translated into \geq and \leq . Besides, Poise sets a default range (0~10000) for comparisons without considering the real range of variables.
- *#5: Missing table entries.* Poise provides the *monitor* expression *count(p)* that counts the number of packets satisfying a predicate p . However, table entries are not generated. Thus, no packets would satisfy the predicate and be counted.
- *#6: Missing action parameters.* Poise uses registers to maintain states. However, for some registers, the read/write actions do not specify the index to read or write.

For Marple, *Firebolt* finds 2 types of intent violations in 23 generated programs out of 7329 generated programs in total.

- *Bug #1: Incorrect infinity translation.* Marple uses *infinity* to represent a variable that exceeds its pre-defined upper limit. However, it assigns a fixed value $2^{31} - 1$ to *infinity* without considering the actual upper bound.
- *Bug #2: Incorrect key storage.* Marple uses 32-bit registers to store keys in the *groupby* query. When storing a value with a width greater than 32 bits, e.g., the ingress timestamp, the stored value would be the truncation of the value.

Table 3: Detected bugs by existing verification tools.

DP Generator Under Test	# Intents	# Detected Bugs		
		Crash Bug	Security Vulnerability	Intent Violation
Marple	14	0	1 (①)	0
Sonata	13	0	2 (①②)	1 (②)
Poise	7	0	2 (①②)	4 (②③④⑥)

Crash bug. *Firebolt* finds one crash bug in Marple, while Sonata and Poise do not report any crash bugs. Marple converts the division expression $(a/2^b)$ into a right-shift operation $(a \gg b)$, but sets the maximum shift width to a fixed value of 8. According to the code, the generator would crash when the exponent b satisfies a legal value of $8 < b < \log_2 a$.

6.2 Bug Coverage

To evaluate the bug coverage of *Firebolt*, first, we compare the bugs detected using different intent space pruning methods to demonstrate that the intent generation approach of *Firebolt* is able to thoroughly cover the intent space to find bugs. Then, we compare the bugs detected between *Firebolt* and existing verification tools to demonstrate that the automatic testing of *Firebolt* can detect more bugs, compared to verifying hand-written test cases using existing tools.

Intent representativeness. We examine whether our two intent space pruning methods (§4.3) compromise intent representativeness by checking the bug coverage of generated intents. For each type of pruning method, we configure the extent of the other method as the default value (the number of random rules $r = 1$ for intra-symbol and the combination factor $n = 2$ for inter-symbol), vary the pruning extent of the current method, and check the resulting bug coverage. As there exist infinite possible intents, obtaining all intents is impractical. We randomly generate 50K intents ($> 10 \times$ intents generated by *Firebolt*) as the baseline.

We present the results in Table 2. The *None* row represents the baseline, where only 6 bugs are discovered (19 by *Firebolt*). This is because a limited number of intents (50K) represent a very small fraction of the entire intent space. Intra-symbol pruning can greatly reduce the intent space, but increasing the number of random rules (r) from 1 to 2 does not increase the bug coverage. For inter-symbol pruning, we can see that a small combination factor ($n = 1$) can find many bugs, but misses one interactive bug, i.e., Sonata’s *reduce-then-filter* bug, which can be found when $n = 2$. Further increasing the combination factor ($n = 3$) cannot find more bugs, but greatly increases generated intents from O(1K) to above 50K.

Therefore, compared to random intent generation, *Firebolt* intent pruning can maintain representativeness with much fewer test cases. Moreover, the recommended pruning extent is $r = 1$ for intra-symbol pruning and $n = 2$ for inter-symbol pruning, which is adequate for comprehensive bug detection.

Comparison with existing tools. We compare the bug coverage of *Firebolt* with that of existing verification tools [30, 31]. For existing verification tools, we collect all open-source

manually-written intents [43–45] as input to find as many bugs as possible. For each collected intent, we manually write the corresponding specification for each verification tool and perform program verification. The results are shown in Table 3. With a limited number of $O(10)$ hand-written intents, the existing verification tools can only discover 10 bugs out of 19 by *Firebolt*, and cannot find other bugs undetected by *Firebolt*. This highlights the high bug coverage of *Firebolt* over existing tools. By delving into these open-source intents, we find that the developers of DP generator did make an effort to write different examples, but the hand-written test cases struggle to efficiently find all bugs in DP generators.

To find more bugs with existing tools, we can use the (1000s of) intents generated by *Firebolt* as input for existing tools. However, doing so requires manually writing specifications for 1000s of intents, which is time-consuming and error-prone. We will analyze the scalability issues in §6.3.

6.3 Efficiency

Next, we evaluate the debugging efficiency of *Firebolt* by counting the lines of input human-written codes (Figure 3), the lines of intents that *Firebolt* generates, the size of generated test cases (including intents, P4 programs, and table entries), and the running time for intent generation and program verification. We summarize the results in Table 1.

Human-written LoC. In general, *Firebolt* requires a limited number of $O(10)$ LoC for intent grammar, $O(10)$ LoC for semantic constraints, and $O(100)$ LoC for per-symbol specifications. Although the per-symbol specifications occupy the majority of human-written LoC, writing specifications is also required for existing verification tools, and *Firebolt* is still the most efficient. We further discuss *Firebolt* scalability in §6.4.

Test case size. As *Firebolt* utilizes pruning mechanisms to generate representative intents, the resulting intents are relatively small, *i.e.*, from one LoC to 10s of LoC. For the same reason, only a few table entries are generated. Marple even has no output table entries, since it uses flexible expressions in the P4 program to implements the intents. Finally, corresponding P4 programs are often with 100s of LoC. This is because generated programs contain many necessary components for all intents such as the definition of headers and parsers. Thus, even the smallest P4 program contains 100s of LoC.

Running time. *Firebolt* generates $O(1K)$ intents for each DP generator. Intent generation and program verification in all scenarios can be done within 25 minutes. DP generators with more semantic constraints (*e.g.*, Marple) take more time to generate a correct intent (23ms vs 3ms for Sonata). This is because relatively more semantically invalid intermediate sentential forms will be detected and rejected during generation.

6.4 Scalability

Manual effort. We compare the manual efforts (*i.e.*, lines of specifications) required by *Firebolt* and existing verification

Table 4: Comparing lines of human-written specifications.

DP Generator Under Test	Verifying One Program	Verifying All Generated Programs	Finding All Bugs (1 Bug / 1 Program)
p4v	$O(1K)$	$O(1M)$	$O(10K)$
Aquila	$O(100)$	$O(100K)$	$O(1K)$
Firebolt	$O(100)$		

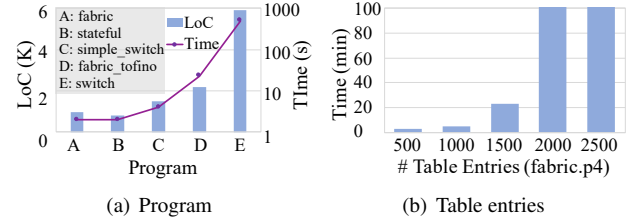


Figure 9: Time needed to verify larger P4 programs with more table entries.

tools, Aquila [31] and p4v [30], to debug a DP generator with equal bug coverage. This means that both *Firebolt* and verification tools take thousands of intents generated by *Firebolt* as input. As shown in Table 4, *Firebolt* requires $O(100)$ of per-symbol LoC to automatically generate the specifications of all intents. In comparison, verification tools require $O(100)$ to $O(1K)$ LoC to convert one intent. Converting all intents means $O(100K)$ to $O(1M)$ LoC. Under equal bug coverage, *Firebolt* consumes merely 0.1% to 0.01% manual efforts compared to existing tools. Moreover, human-written specifications can be faulty, which further reflects the scalability of *Firebolt*.

Scaling to larger test case. We evaluate the scalability of *Firebolt* when verifying larger P4 programs with more table entries. We use several open-source or vendor-supplied P4 programs instead of the small programs generated by *Firebolt*.

First, we measure the time required by *Firebolt* to verify P4 programs of different sizes. For each program, we manually write <3 entries for each table, and also write the corresponding specifications for verification. As shown in Figure 9(a), *Firebolt* requires more time for verification as the complexity of the P4 program increases. Nevertheless, even the most complex switch.p4 can be verified in 8 minutes.

Next, we compare the verification time when installing different numbers of table entries to the same P4 program fabric.p4. As shown in Figure 9(b), the number of table entries has a larger impact on the verification time than the size of P4 program. When the number of entries does not exceed 1500, *Firebolt* can complete the verification in <30 minutes. When the number of entries exceeds 2000, *Firebolt* takes >100 minutes for verification. The increase is not linear because the table entries are converted into if-then-else branches, resulting in an exponential increase in the size of the generated Z3 formulas. This non-linear scalability of verification time with table entries has also been recognized in other verification works [31] and solved using encoding optimizations. Currently, *Firebolt* by design generates small test cases without losing bug coverage using intent space pruning. For upcoming generators, we may encounter larger intents, programs, and table entries that become more time-consuming

to verify. In that case, *Firebolt* can refer to the optimization techniques in existing verification tools [30, 31] to accelerate the verification of individual intent-program pairs.

7 Discussion

Human effort required by *Firebolt*. *Firebolt* requires three inputs, including grammar, semantic constraints, and per-symbol specification to debug a DP generator. First, the grammar should already be provided by the designers of DP generators [14, 16, 21–23] so that the DP generator can be used correctly by others. Second, accurate semantic constraints also help to better use the DP generator. We summarize the semantic constraints of three advanced DP generators in Appendix A. They all have <20 semantic constraints that can be classified into four types, *i.e.*, banned variable redefinition, necessary variable definition, illegal variable reference, and special ones. The former three types account for the majority of the constraints and are closely related to variable definitions and references. Semantic constraints can shrink the intent space, and missing some semantic constraints will not affect the bug coverage but merely produce more intents. Third, *Firebolt* requires manually writing per-symbol specification. However, compared with existing tools, *Firebolt* saves significant human efforts by automatically composing per-symbol specifications into intent specifications.

Cross-platform generality of *Firebolt*. The formalization phase of *Firebolt* considers extern behaviors because they are critical for the correctness of DP generators. However, the semantics of extern behaviors are target-specific. Currently, *Firebolt* supports two common extern implementations including stateful memory and hash calculation. Since externs can be taken as arithmetic operations on some variables (*e.g.*, temporal variable, metadata field, and packet header), they can always be converted into logical Z3 formulas. As a future work, we would like to extend *Firebolt* to support user-defined extern semantics to improve cross-platform generality.

8 Related Work

Data plane generator. To simplify DP programming, a growing body of research proposes data plane generators which convert high-level intents into platform-specific DP programs. DP generators provide primitives to specify developer intents in different domains, *e.g.*, query primitives for monitoring tasks [15–20], measurement and control primitives to specify security policies [21, 22] and routing policies [23], and some other intent languages for their own purposes [14, 24–27]. DP generators greatly relieve the burden of DP programming, but their own correctness is not guaranteed. In this paper, we design a blackbox-based testing system to debug them.

Data plane program verifier. Several efforts have been proposed to verify DP programs. P4-assert [28] and Vera [29] translate P4 programs into other language models (SEFL

and C) and rely on existing symbolic execution framework (SymNet [46] and Klee [47]) to analyze the behavior of the resulting programs. p4v [30] and Aquila [31] use Dijkstra’s classic verification approach by formalizing the P4 program in Guarded Command Language (GCL) and using the Z3 theorem prover [35] to check whether the specifications hold. Some other tools, such as bf4 [32] and P6 [34], utilize various techniques such as static verification, code changes and runtime checking to ensure that the deployed P4 program is bug-free. However, using these tools to debug DP generators cannot cover all generator bugs and requires massive manual efforts to verify each possible intent-program pair. Different from all of them, *Firebolt* can automatically generate representative intents as test cases for high coverage and automatically produce intent specifications for high scalability.

Testing in networking. Testing is a popular technique to find bugs in network systems by generating and running many test cases. Metha [48] tests network verification tools by generating network configurations as test cases and comparing the tool’s output with that of the actual router. p4pktgen [49] tests P4 programs by generating test cases using symbolic execution. Gauntlet [42] and P4Fuzz [50] both test the P4 compiler by generating random P4 programs. If the intermediate representation (IR) of the compiler is accessible, Gauntlet compares the transformed programs after different compiler passes, otherwise it generates packets to test the behavior of the P4 program to debug the compiler. P4Fuzz compares the output of different compilers to find potential bugs. *Firebolt* also uses generation-based testing to debug DP generators. However, unlike existing work to test specific targets, *Firebolt* needs to handle a variety of DP generators. *Firebolt* adopts a syntax-guided approach to generate test cases and designs generic methods to verify the correctness of each test case.

9 Conclusion

This paper presents *Firebolt*, a blackbox testing tool designed to debug DP generators. We propose syntax-guided intent generation with semantic constraint injection and intent space pruning techniques, and program verification with automatic intent and program formalization. By evaluating three popular open-source DP generators, we show the high bug coverage and scalability of *Firebolt* compared to existing solutions.

This work does not raise any ethical issues.

Acknowledgement

We sincerely thank our shepherd, Fernando Ramos, and the anonymous reviewers for their constructive comments. Ying Liu and Chen Sun are the corresponding authors. This work is supported by National Key R&D Program of China (Grant No. 2018YFB1800405), National Natural Science Foundation of China (Grant No. 61772307), and Beijing Natural Science Foundation (Grant No.4222026).

References

- [1] Barefoot Networks. Tofino. Website, 2019. <https://www.barefootnetworks.com/products/brief-tofino/>.
- [2] Cavium. Xpliant ethernet switch product family. Website. <https://www.cavium.com/xpliant-ethernet-switch-product-family.html>.
- [3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.
- [4] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rotenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176, 2017.
- [5] Qun Huang, Patrick PC Lee, and Yungang Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 576–590, 2018.
- [6] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–16, 2018.
- [7] Joel Hypolite, John Sonchack, Shlomo Hershkop, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. Deepmatch: practical deep packet inspection in the data plane using network processors. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 336–350, 2020.
- [8] Yehuda Afek, Anat Bremler-Barr, and Lior Shafir. Network anti-spoofing with sdn data plane. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.
- [9] Jiamin Cao, Ying Liu, Yu Zhou, Chen Sun, Yangyang Wang, and Jun Bi. Cofilter: A high-performance switch-accelerated stateful packet filter for bare-metal servers. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2019.
- [10] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 161–176, 2019.
- [11] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, 2019.
- [12] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Packet subscriptions for programmable asics. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 176–183, 2018.
- [13] The P4.org language consortium. P4_16 reference compiler. <https://github.com/p4lang/p4c>.
- [14] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 435–450. ACM, 2020.
- [15] Yu Zhou, Jun Bi, Tong Yang, Kai Gao, Jiamin Cao, Dai Zhang, Yangyang Wang, and Cheng Zhang. Hypersight: Towards scalable, high-coverage, and dynamic network monitoring queries. *IEEE Journal on Selected Areas in Communications*, 38(6):1147–1160, 2020.
- [16] Vikram Nathan, Srinivas Narayana, Anirudh Sivaraman, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Demonstration of the marple system for network performance monitoring. In *Posters and Demos Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 57–59. ACM, 2017.
- [17] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: query-driven streaming network telemetry. In Sergey Gorinsky and János Tapolcai, editors, *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, pages 357–371. ACM, 2018.

- [18] Ross Teixeira, Rob Harrison, Arpit Gupta, and Jennifer Rexford. Packetscope: Monitoring the packet lifecycle inside a switch. In Proceedings of the Symposium on SDN Research, pages 76–82, 2020.
- [19] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. Newton: intent-driven network traffic monitoring. In Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies, pages 295–308, 2020.
- [20] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative network monitoring with netqre. In Proceedings of the conference of the ACM special interest group on data communication, pages 99–112, 2017.
- [21] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. Programmable in-network security for context-aware BYOD policies. In 29th USENIX Security Symposium (USENIX Security 20), pages 595–612, 2020.
- [22] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qi Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020. The Internet Society, 2020.
- [23] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A programmable system for performance-aware routing. In Ranjita Bhagwan and George Porter, editors, 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020, pages 701–721. USENIX Association, 2020.
- [24] Jiamin Cao, Yu Zhou, Ying Liu, Mingwei Xu, and Yongkai Zhou. Turbonet: Faithfully emulating networks with programmable switches. In 2020 IEEE 28th International Conference on Network Protocols (ICNP), pages 1–11. IEEE, 2020.
- [25] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Packet subscriptions for programmable asics. In Proceedings of the 17th ACM Workshop on Hot Topics in Networks, pages 176–183, 2018.
- [26] Yu Zhou, Zhaowei Xi, Dai Zhang, Yangyang Wang, Jinqiu Wang, Mingwei Xu, and Jianping Wu. Hypertester: high-performance network testing driven by programmable switches. In Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies, pages 30–43, 2019.
- [27] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, pages 296–309, 2020.
- [28] Miguel C. Neves, Lucas Freire, Alberto E. Schaeffer Filho, and Marinho P. Barcellos. Verification of P4 programs in feasible time using assertions. In Xenofontas A. Dimitropoulos, Alberto Dainotti, Laurent Vanbever, and Theophilus Benson, editors, Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018, pages 73–85. ACM, 2018.
- [29] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 programs with vera. In Sergey Gorinsky and János Tapolcai, editors, Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018, pages 518–532. ACM, 2018.
- [30] Jed Liu, William T. Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Calin Cascaval, Nick McKeown, and Nate Foster. p4v: practical verification for programmable data planes. In Sergey Gorinsky and János Tapolcai, editors, Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018, pages 490–503. ACM, 2018.
- [31] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, Jie Lu, Xionglie Wei, Hongqiang Harry Liu, Ming Zhang, Chen Tian, and Minlan Yu. Aquila: a practically usable verification system for production-scale programmable data planes. In Fernando A. Kuipers and Matthew C. Caesar, editors, ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021, pages 17–32. ACM, 2021.
- [32] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. bf4: towards bug-free p4 programs. In Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, pages 571–585, 2020.
- [33] Nuno P Lopes, Nikolaj Bjørner, Nick McKeown, Andrey Rybalchenko, Dan Talayco, and George Varghese. Automatically verifying reachability and well-formedness in p4 networks. Technical Report, Tech. Rep., 2016.

- [34] Apoorv Shukla, Kevin Hudemann, Zsolt Vági, Lily Hügerich, Georgios Smaragdakis, Artur Hecker, Stefan Schmid, and Anja Feldmann. Fix with p6: Verifying programmable switches at runtime. In IEEE INFOCOM 2021 - IEEE Conference on Computer Communications, pages 1–10, 2021.
- [35] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In International conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 337–340. Springer, 2008.
- [36] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, page 436–449, New York, NY, USA, 2018. Association for Computing Machinery.
- [37] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013, pages 1–8. IEEE, 2013.
- [38] Donald E Knuth. Backus normal form vs. backus naur form. Communications of the ACM, 7(12):735–736, 1964.
- [39] Armin Cremers and Seymour Ginsburg. Context-free grammar forms. Journal of Computer and System Sciences, 11(1):86–117, 1975.
- [40] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O’Neill. A grammar design pattern for arbitrary program synthesis problems in genetic programming. In James McDermott, Mauro Castelli, Lukás Sekanina, Evert Haasdijk, and Pablo García-Sánchez, editors, Genetic Programming - 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings, volume 10196 of Lecture Notes in Computer Science, pages 262–277, 2017.
- [41] Changhai Nie and Hareton Leung. A survey of combinatorial testing. ACM Comput. Surv., 43(2):11:1–11:29, 2011.
- [42] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. Gauntlet: Finding bugs in compilers for programmable packet processing. In 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020, pages 683–699. USENIX Association, 2020.
- [43] Marple code. <https://github.com/performance-queries/marple>.
- [44] Sonata code. <https://github.com/Sonata-Princeton/SONATA-DEV>.
- [45] Poise code. <https://github.com/qiaokang92/poise>.
- [46] Radu Stoescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In Proceedings of the 2016 ACM SIGCOMM Conference, pages 314–327, 2016.
- [47] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI, volume 8, pages 209–224, 2008.
- [48] Rüdiger Birkner, Tobias Brodmann, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Metha: Network verifiers need to be correct too! In James Mickens and Renata Teixeira, editors, 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021, pages 99–113. USENIX Association, 2021.
- [49] Andres Nötzli, Jehadad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. P4pktgen: Automated test case generation for p4 programs. In Proceedings of the Symposium on SDN Research, pages 1–7, 2018.
- [50] Andrei-Alexandru Agape, Madalin Claudiu Danceanu, Rene Rydhof Hansen, and Stefan Schmid. P4fuzz: Compiler fuzzer for dependable programmable dataplanes. In International Conference on Distributed Computing and Networking 2021, pages 16–25, 2021.

Appendix A Semantic Constraints of Advanced DP Program Generators

Table 5 lists the identified semantic constraints for Marple [16], Sonata [17], and Poise [21]. The semantic constraints can be classified into four categories, *i.e.*, banned variable redefinition, necessary variable definition, illegal variable reference to generate *compilable* intents, and other special constraints to generate *complete* intents. For each category, we give an example of how the constraint can be expressed with the formal *if-then* expressions in §4.2.

Table 5: Summary of semantic constraints of Marple [16], Sonata [17], and Poise [21].

Constraints		Description	Type	Expression
Marple	Banned Variable Redefinition (3)	#1: Each query has a stream name, which cannot be repeatedly defined.	Exclusion	#1 as an example: $if \exists r_1 \text{ on } n_1, \nexists r_2 \text{ on } n_2, n_1 \leftrightarrow n_2$ $\langle n_1 \rangle: *, \langle prog \rangle \langle streamStmt \rangle \langle streamName \rangle, *$ $\langle r_1 \rangle: \langle streamName \rangle \rightarrow *$ $\langle n_2 \rangle: *, \langle prog \rangle \langle streamStmt \rangle \langle streamName \rangle, *$ $\langle r_2 \rangle: \langle r_1 \rangle$
		#2: Each aggregation function has a function name, which cannot be repeatedly defined.		
		#3: A aggregation function may define multiple aggregation states. The aggregation state names cannot be repeatedly defined.		
	Necessary Variable Definition (8)	#4~7: Each query (<i>map/groupby/filter/zip</i>) operates on a stream, which should be either defined or the default input stream T.	Dependency	#4 as an example: $if \exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \rightarrow n_2$ $\langle n_1 \rangle: *, \langle prog \rangle \langle streamStmt \rangle \langle streamQuery \rangle \langle map \rangle \langle streamName \rangle, *$ $\langle r_1 \rangle: \langle streamName \rangle \nrightarrow T$ $\langle n_2 \rangle: *, \langle prog \rangle \langle streamStmt \rangle \langle streamName \rangle, *$ $\langle r_2 \rangle: \langle r_1 \rangle$
		#8: <i>groupby</i> queries take an aggregation function name as input. The function should be defined.		
		#9: <i>groupby</i> queries may include self-defined variables in the aggregation key. The variables should be defined.		
#10: <i>filter</i> queries may reference self-defined variables in its predicate. The variables should be defined.				
Illegal Variable Reference (2)	#11: <i>map</i> queries may reference self-defined variables. The variables should be defined.	Exclusion	#12 as an example: $if \exists r_1 \text{ on } n_1, \nexists r_2 \text{ on } n_2, n_1 \leftrightarrow n_2$ $\langle n_1 \rangle: *, \langle prog \rangle \langle streamStmt \rangle \langle streamQuery \rangle \langle map \rangle \langle map_col \rangle, *$ $\langle r_1 \rangle: \langle map_col \rangle \rightarrow *$ $\langle n_2 \rangle: *, \langle prog \rangle \langle streamStmt \rangle \langle streamQuery \rangle \langle map \rangle \langle map_col \rangle, *$ $\langle r_2 \rangle: \langle r_1 \rangle$	
	#12: <i>map</i> queries assign specified or self-defined variables to computed expressions, where the variables should not be assigned repeatedly.			
Special Semantic Constraints (1)	#13: <i>zip</i> queries merge fields in <i>different</i> streams.	Exclusion	#14 as an example: $if \exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \rightarrow n_2$ $\langle n_1 \rangle: *, \langle prog \rangle \langle streamStmt \rangle \langle streamQuery \rangle, *$ $\langle r_1 \rangle: \langle streamQuery \rangle \rightarrow \langle map \rangle$ $\langle n_2 \rangle: *, \langle prog \rangle \langle streamStmt \rangle \langle streamName \rangle, *$ $\langle r_2 \rangle: \langle streamQuery \rangle \rightarrow \langle filter \rangle \langle groupby \rangle \langle zip \rangle$	
	#14: An intent should not end with a <i>map</i> query.			
Sonata	Special Semantic Constraints (2)	#1: An intent should not end with a <i>map</i> query.	Dependency	#1 as an example: $if \exists r_1 \text{ on } n_1, \exists r_2 \text{ on } n_2, n_1 \leftarrow n_2$ $\langle n_1 \rangle: *, \langle prog \rangle \langle streamQuery \rangle, *$ $\langle r_1 \rangle: \langle streamQuery \rangle \rightarrow \langle map \rangle$ $\langle n_2 \rangle: *, \langle prog \rangle \langle streamQuery \rangle, *$ $\langle r_2 \rangle: \langle streamQuery \rangle \rightarrow \langle filter \rangle \langle reduce \rangle \langle distinct \rangle$
		#2: <i>reduce</i> queries should follow <i>map</i> queries.		
Poise	Banned Variable Redefinition (2)	#1: Each <i>list</i> has a name, which cannot be repeatedly defined.	Exclusion	Similar to the banned variable redefinition of Marple.
		#2: Each <i>monitor</i> function has a name, which cannot be repeatedly defined.		
	Necessary Variable Definition (2)	#3: Each <i>monitor</i> function references a <i>list</i> , which should be defined.	Dependency	Similar to the necessary variable variable definition of Marple.
		#4: A <i>statement</i> may reference a <i>monitor</i> function, which should be defined.		



Investigating Managed Language Runtime Performance

Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be faster?

David Lion^{*†}, Adrian Chiu^{*}, Michael Stumm^{*}, Ding Yuan^{*†}
^{*}University of Toronto, [†]YScope

Abstract

The most widely used programming languages today are managed languages. They are popular because their vast features improve many aspects of code development, including increased productivity and safety. However, as a product or service scales in usage, performance issues become a problem. Developers are then often faced with complex choices as they must decide whether the desired performance can be squeezed from existing code, or whether their language has reached its performance limits, requiring years of code to be ported to a new more-performant language. To make matters worse, runtime performance is shrouded in mystery as it involves complex interactions of different components, such as interpreter, just-in-time (JIT) compiler, thread library, and Garbage Collection (GC) system.

We present an in-depth performance analysis and comparison of Java, Go, JavaScript, and Python, using C++ as a baseline. We carefully instrumented the different language runtimes, so that developers can precisely measure the number of cycles taken to execute any bytecode instruction, or the overhead of dynamic type checking in JavaScript. This allows us to accurately identify sources of overhead. We further created 6 applications from the ground up to establish the *LangBench* benchmark; the applications cover a range of complexity, and they cover a variety of application scenarios differing in compute intensity, memory usage, network and disk I/O intensity, and available concurrency. We comprehensively analyze their completion times, resource usage, and scalability.

Overall, we found that V8/Node.js and CPython exhibit excessive overheads, executing applications 8.01x and 29.50x slower on average than their C++ counterparts, respectively. Making matters worse, applications on these two runtimes scale poorly in that they cannot effectively utilize more than one core. In contrast, OpenJDK and Go applications are performance competitive to C++, running only 1.43x and 1.30x slower, respectively, and they can easily scale to multiple cores. There are applications where OpenJDK and Go outperform their C++ counterparts.

1 Introduction

Programming languages with integrated run-time environments have continuously grown in popularity. The three most popular languages on GitHub since 2015 are JavaScript, Java, and Python [38]. These languages offer the promise of improved developer productivity and thus faster product creation and adaptation because of a variety of features they offer, including easier readability and usability, dynamic type checking, memory management with garbage collection, and dynamic memory safety checks. We use the term “*managed languages*” to refer to these type of programming languages.

Managed languages are increasingly being used to implement *systems software* where performance is critical. Both Hadoop [54] and Spark [76] run on a Java Virtual Machine (JVM) [61] as they are implemented in Java and Scala respectively. Kubernetes [21], etcd (a distributed key-value store [4]), and M3 (a distributed time series database and query engine built by Uber [14]) are all implemented in Go. Recently, even an operating system (OS) kernel, Biscuit [52], was implemented in Go [8]. Openstack [18], Paypal [1], Instagram [22], and Dropbox all heavily utilize Python; Python is Dropbox’s “most widely used language both for backend services and the desktop client app” with almost 4 million lines of Python code in one repository [20]. As a final example, JavaScript is used in the performance critical path for the Bladerunner pub/sub system at Facebook [49].

Several factors come into play when selecting a programming language for a new service, including current developer expertise and experience, constraints imposed by existing ecosystems (e.g., home-grown libraries, development, debugging tools, performance monitoring and logging systems, etc.), and developer productivity. Managed languages are an attractive proposition, precisely because they offer the promise of higher developer productivity, leading to faster project completion times. The performance of the language is rarely a consideration at the outset, in part because of the belief that performance issues can be addressed later, perhaps through horizontal scaling by simply adding hardware. Some go as far as to claim “*Choosing a language for your application*

simply because it's 'fast' is the ultimate form of premature optimization" [47].

However, performance will ultimately become a priority as the usage of the service begins to scale and the service becomes too slow or the cost of hardware becomes too high. Developers then begin a large sequence of performance optimizations that can grow into herculean efforts. For example, Twitch.tv and others use tricks and tweaked parameters to meet desired GC performance in Go [5, 19]; project Tungsten in Spark goes as far as to bypass the JVM, to squeeze out performance [23].

But there can come a point where incremental optimizations (requiring much time and effort) no longer suffice and a more radical solution must be considered, namely switching to a "better performing" language. A few examples from industry: Stream abandoned Python for Go, as Python would spend 10ms creating objects from data that Cassandra took 1ms to fetch, noting that "We've been optimizing Cassandra, PostgreSQL, Redis, etc. for years, but eventually, you reach the limits of the language you're using." Discord switched from Go to Rust claiming that "Rust was able to outperform the hyper hand-tuned Go version." [43]. Performance issues are also cited as the main reason Twitter was forced to switch from Ruby on Rails to Scala and Java [40, 42].

When selecting a new language for performance reasons, the question is: what language? Understanding the performance and scalability implications of a (new) language today is non-trivial, especially for managed languages. This is for several reasons.

First, no empirical studies exist that scientifically compare the different managed languages. The primary source of information available today is the blogosphere containing heated "religious" debates that include tunnel-visioned anecdotes with few rigorous measurements to back up stated claims. For example, while many believe programs written in Java run slower than when written in C/C++ [27], others suggest that Java programs can be faster than C, because the JIT compiler produces faster machine code by leveraging a runtime profile [26]. Similarly, there have been polarized debates with respect to the performance of JavaScript [11, 39], Go [43, 46] and even Python – for example, sources from Paypal claimed that Python offered superior performance over other languages and reported multiple cases where Python outperformed their C++ and Java counterparts while requiring less code [1]. Similarly, developers reported that Python could outperform both C/C++ and Java when using regular expressions or strings [33–35].

Discussions on *scalability* are even muddier. For example, in a popular blog by the official Node.js Medium account, developers conclude that by being event-driven and asynchronous, JavaScript is *ideal* for scaling to millions of concurrent connections, despite its event loop only executing on a single thread [45]. As another example, while it should be well-known that CPython, the de facto runtime for Python today, uses a global interpreter lock (GIL) that will serialize

all concurrent thread executions, Paypal's engineering blog claims that it scales well, and noted that "Dropbox, Disqus, Eventbrite, Reddit, Twilio, Instagram, Yelp, EVE Online, Second Life, and, yes, eBay and PayPal all have Python scaling stories that prove scale is more than just possible: it's a pattern" [1].

Second, no benchmark suite is publicly available today that enables a meaningful comparison between different managed languages (and their implementations). Existing benchmark suites target specific languages or applications. Extending these benchmarks to other languages is often impossible; for example, the DaCapo benchmark for Java contains applications such as Eclipse, a full-featured IDE [50]. As a result, any comparison on language runtime performance often compares apples to oranges.

Third, language runtime systems are extremely complex software systems, providing multiple abstractions that all affect performance. For example, a developer must understand the interpreter, possibly multiple JIT compilers (e.g., the OpenJDK JVM contains 4 levels of JIT compilation), a memory management subsystem that performs garbage collect, the behavior of thread libraries, etc.

Finally, there are no helpful, publicly available profiling tools for understanding the overheads of language runtime systems. The language subsystems themselves expose little profiling information on their internals. For example, while it is widely speculated that dynamic type checking adds significant overhead [44], V8/Node.js does not expose any performance counters to report this overhead.

In this paper, we present an in-depth quantitative performance analysis of four of the popular managed languages with their most widely used runtime systems: CPython, OpenJDK, Node.js with the V8 engine for JavaScript, and the reference Go compiler [15, 24, 36, 38]. We compare their performance characteristics with that of C++ on GCC as the baseline. Our focus is primarily on understanding their differences with respect to speed and scalability.¹ We chose these languages not just because of their popularity, but because they represent different designs along the following three dimensions:

- **Typing.** JavaScript and Python are dynamically typed, meaning the runtime must determine the type of objects at run time, whereas others are statically typed. This allows us to understand the performance impact of dynamic type checking.
- **Execution modes.** Only Go is ahead-of-time compiled. The other runtimes first interpret bytecode, and compile hot functions Just-In-Time (JIT). The exception is CPython that only has an interpreter and no JIT compiler. This enables us to compare three execution modes: native-only (Go and

¹An analysis of resource usage is left to the Appendix, because the findings have largely already been established by prior studies [63], and we did not need to implement any additional profiling mechanisms.

C++), a combination of interpreter and JIT-compiled native (OpenJDK and V8), and interpreter only (CPython).

- **Concurrency models.** V8/Node.js is event driven where event handlers are executed sequentially on a single kernel thread. Similarly, CPython has a global interpreter lock (GIL) so only one thread can execute Python code at a time. Go has its own scheduler and provides *user threads* as “goroutines.” Its scheduler decides how many kernel threads to use for the developer’s goroutines. OpenJDK’s Thread is simply a kernel thread.

Contributions

The contributions of the paper are as follows:

Runtime instrumentations. We are the first to make publicly available (as artifacts) instrumentations for the three runtime systems of popular managed languages that are not statically compiled, namely OpenJDK, V8, and CPython. Implementing such instrumentations is challenging given the complexity of these runtimes and the fact that two of them are implemented in assembly and IR. Our instrumentations enable bytecode-level profiling of (1) the execution overhead of any target *bytecode* in the interpreter and (2) the dynamic type-checking overhead in Node.js/V8. The profiling information generated, in turn, can be used to guide optimization efforts at the application level and can enable effectual optimizations. The instrumentations are described in §2.

Benchmark suite. We are the first to make publicly available (as artifacts) six applications suitable for evaluating managed languages; these applications were used to create twelve benchmarks. The applications, which range from micro-benchmarks to real applications, cover a variety of scenarios, differing in compute intensity, memory usage, I/O intensity, relative startup time, and the degree of available concurrency. In particular, we took care to expose the differences in the three major design dimensions mentioned above. Three of the six applications are parallel, and we parallelize them using both multithreading and multiprocessing where applicable. The benchmark suite is called **LangBench** and is described in §3. The source code of our instrumented runtimes and LangBench can be found at <https://github.com/topics/langbench>.

Comparative analysis. We quantitatively analyse the performance of the benchmarks in our suite and identify how the individual runtimes improve or hinder performance relative to the respective C++ implementations. Our objective was to compare the runtimes of the target managed languages in an objective, scientific way. Many of our results are not particularly surprising (even if they contradict some views held in the blogosphere). For example, Go and OpenJDK perform significantly better than V8/Node.js and CPython, with CPython performing worst by far, even when compared against V8 and OpenJDK’s interpreter-only execution (§6). CPython and

V8/Node.js do not benefit from parallelism; in fact, increasing the number of threads systematically decreases performance (§7). A major source of V8’s relatively poor performance is its dynamic type checking, even when the JS code only uses primitive types that never change (§6.1).

Perhaps more surprising is the fact that in many cases, the abstractions offered by runtimes can actually lead to speedups over GCC (§8). This contradicts the conventional wisdom that abstraction comes at the expense of performance [74]. OpenJDK outperformed GCC in three of the benchmarks, because the moving garbage collector actually improves cache locality. This leads to the unintuitive behavior that the more frequently GC is performed, the *better* the overall performance. Go abstracts away the usage of kernel threads, reducing the number of context switches and kernel threads. Finally, abstracting away low-level I/O operations allows runtimes to use optimal I/O system call configurations, outperforming the idiomatic approach in C++.

Limitations

The main limitation of our work is that it does not, and cannot, comprehensively answer every question one might have related to the performance of a language runtime. We only evaluated the runtimes of four languages, and for each language we only evaluated the implementation that is the most widely used. In addition, we only ran our workloads on a single OS/hardware stack. Our findings pertain to our benchmarks, which model real-life applications, but may not be representative of a vast range of applications. Accordingly, our study is not meant to determine the best or most performant programming language for any particular application. Furthermore, our benchmark and analysis do not focus on some performance aspects. Notably, we do not study the overhead of garbage collection when under memory pressure (there is a gap between the working set size of our benchmarks and the maximum heap size setting). There is a large body of prior work focusing on this aspect already [51, 63, 79]. Similarly, our benchmark is not meant to measure the various JIT compiler’s optimizations, as there are also a large number of existing benchmarks meant to do exactly that [10, 32, 53].

2 Language Runtime Instrumentation

In this section we describe how we instrumented the three runtimes: OpenJDK’s HotSpot JVM, Node.js/V8, and CPython. Our instrumentations measure two types of information: (1) the performance of the execution of any bytecode instruction in the interpreter, and (2) the dynamic type and bounds checking overhead in V8’s JIT compiled code. Users can specify a bytecode instruction to measure its overhead, or any JavaScript (JS) function to measure the type and bounds checking overhead when executing that function.

Why profile interpreter performance? Some have the view that interpreter performance is not important as it mostly affects the startup time, which will be amortized by “warm execution.” We do not share this view. While interpreter performance may have been irrelevant over a decade ago when workloads ran in large, long-running monolithic applications that handle all requests [75], the paradigm shift to the cloud [65, 69, 77] and data analytics [62] expose the runtime’s startup performance as being significant. For example, auto-scaling in the cloud often results in the bringing up of additional instances in the face of a load spike [65, 69]; the problem is also exemplified by short-running instances in Function-as-a-Service platforms [65, 77]. In 2020, the median AWS Lambda invocation ran for only *60 milliseconds* [37], while startup times for the JVM and V8 are on the order of hundreds of milliseconds or even seconds [62, 65, 77, 80]. Similarly, data analytics systems face a fundamental tension between parallelizing long running jobs into shorter tasks and the runtime’s start-up overhead [62].

In practice, instead of ignoring the performance of the interpreter, implementers spend huge efforts in optimizing the interpreter. For example, OpenJDK has two interpreter implementations: one in C++ and the other entirely in hand-crafted x86 assembly; in one benchmark (§6.2), we found that the C++ interpreter to be 1.93x slower than the assembly one, which is perhaps why the C++ interpreter is only used on non-x86 platforms. Similarly, V8’s interpreter is written in hand-crafted IR, and IBM’s OpenJ9 Java runtime has significant optimizations targetting startup time which is featured as a major advantage over OpenJDK in the cloud [55, 69, 75].

Bytecode-level profiling can guide optimization efforts and can enable effectual optimizations. For instance, developers can optimize their programs to avoid the use of bytecode instructions with high overheads. Instagram engineers did just this by instrumenting CPython to identify the bytecode instructions with high overheads, and then optimizing their code to avoid using these expensive instructions [22]. Bytecode-level profiling also allows us to understand the performance difference between different runtimes.

Why profile type and bounds checking? As we will show in §6.1, dynamic type and bounds checking is a major source of V8’s overhead. Similar to bytecode profiling, programmers can optimize their JS programs to avoid such overhead once the source is identified (§6.1). Our instrumentation also enables eliminating type and bounds checking overhead entirely for those functions where developers know that they are safe. For instance, say a JS function accesses `a[i]`, the element at index `i` of array `a`, and their types never change (known as “monotype”). V8 detects that `a` and `i` are monotype, and it speculatively compiles the function: it checks `a` against the array type (instead of other types) and `i` against integer, before accessing `a[i]`.² But to ensure safety, it cannot remove the

checks because their types could dynamically change in the future. In that case, the check will fail, forcing the JIT-compiled function to exit and be destroyed, and V8 will re-execute the function in its interpreter before recompiling it.

By disabling the checking logic in V8’s JIT compiler for any user-specified JS function, we effectively create a significantly more efficient, albeit unsafe, version of the function. In the above example, developers could enable this feature to turn off the checks when they know `a` and `i` are monotype, so the JIT-compiled code will directly access `a[i]` by indexing into `a` without any checks (effectively turning the JS function into a C function). The difference in execution time of applications with and without checked functions can be significant: e.g., in LangBench’s sort benchmark, disabling the checking in V8 results in 8x speedup (§6.1).

Note that we only instrumented V8’s JIT compiler for identifying type and bounds checking overheads, but not its interpreter. This is because unlike the JIT compiler that independently compiles the different functions, the interpreter’s checking logic is applied to all functions equally, leaving us only with the option of either performing checks in all of an application’s functions or none of them. The latter option would likely to be too risky to be useful in practice. We further note that the checking overhead in the interpreter also becomes negligible when compared to the other overhead from the interpreter, whereas their proportion become much more significant in JIT-compiled code.

Instrumentation Implementation. Conceptually the instrumentations we use to profile bytecode execution in the interpreters are simple. We locate the code block in each interpreter that processes a bytecode instruction, and inject instrumentation around it to collect metrics from the x86 CPU performance counters. In practice, however, adding instrumentation is challenging. One challenge is the complexity of the runtimes: JVM, V8, and CPython consist of approximately 1.2M, 1.0M, and 0.9M lines of code respectively, with little documentation. Instrumenting JVM and V8 is even more challenging as their interpreters are not programmed in a high-level language (e.g., C++) as the other runtimes are, but are generated dynamically at startup time.

The HotSpot JVM has two interpreters. Its default interpreter for x86 is written in hand-crafted assembly (known as the “assembly interpreter”). It also has a interpreter written in C++. We instrumented both. Instrumenting the assembly interpreter brings three challenges. First, one needs to locate the code blocks that process the different bytecodes by searching the assembly code. Second, one has to carefully ensure that the instrumented code does not clobber any registers that are used by the interpreter’s logic. Finally, HotSpot writes the assembly instructions of the interpreter into memory when it starts up and then jumps to the memory location of the beginning of the interpreter. Hence, we need to use the same mechanism in order to be able to embed our instrumentation logic (written in assembly) into memory.

²It also performs other checks as described in §6.1.

```

1  push rax
2  push rcx
3  push rdx
4  rdtscp      ; saves tsc into EDX and EAX registers
5  shlq rdx,32 ; shift tsc's higher 32 bits up in rdx
6  orq rax,rdx ; or onto rax
7  movq dst,rax ; output to a scratch register dst
8  pop rdx
9  pop rcx
10 pop rax

```

Figure 1: The sequence of assembly instructions inlined into the processing of each bytecode instruction.

Figure 1 shows the instruction sequence we inject as part of our instrumentation to obtain the CPU’s timestamp counter (tsc). Line 1-3 saves the registers values.³ The `rdtscp` instruction saves the higher and lower 32 bits of tsc into EDX and EAX respectively, i.e., the lower 32 bits of RDX and RAX [68]. It also clears the higher 32 bits of RDX and RAX. Line 5 shifts the higher 32 bits of tsc, stored in EDX, to the higher 32 bits of RDX, and line 6 effectively concatenates the higher and lower 32-bits of tsc, and stores it into RAX. We embed this instruction sequence at both the beginning and the end of the processing of the target bytecode instruction, so that we can measure the latency by computing the difference.⁴ Similarly, we use `rddpmc` to read other performance counters, including those for cycle and instruction counts.

Instrumenting V8 is even more challenging. V8’s interpreter is written in hand-crafted intermediate representation (IR). When the runtime starts up, the interpreter’s binary is generated dynamically from this IR by the same JIT compiler used at run time in V8. This required us to instrument both the IR code of the interpreter and the JIT compiler so that native instrumentation code is injected correctly.

Specifically, for each target bytecode, we had to locate its processing logic in the interpreter’s IR and then add a new type of IR node we introduced. We further had to modify the JIT compiler, so that when it encounters this new IR node, it produces the correct assembly instructions that collects the CPU performance counters. This was challenging because there is little documentation describing the internals of V8’s interpreter IR or JIT compiler.

One advantage with JVM and V8 is that developers do not need to recompile the runtime when they wish to profile a different bytecode instruction, but only need to restart the runtime. This is because the interpreters are generated dynamically at startup time. Accordingly, we identify which bytecode instruction is to be instrumented at startup time, generate the appropriate instrumentation code so that it is

³We have to manually save the registers because we directly inject code into the assembly code, in contrast to injecting assembly code into C where the `__asm__` block saves the registers.

⁴Intel allows tsc to be synchronized across multicore [48], and Linux enables this synchronization [13]. This ensures a meaningful counter value even if the interpreter thread is migrated to another core during the processing of a bytecode instruction. In reality, however, migration is rare given the processing of bytecode instruction typically only takes tens of cycles.

embedded in the interpreter when written to memory (as with JVM) or generated by the JIT compiler (as with V8).

Instrumenting CPython is far more straightforward because it is written in C++. However, the CPython runtime will have to be recompiled whenever profiling is to be enabled or the target bytecode instruction that should be instrumented is changed. In theory, one could, before the execution of each bytecode, check whether the bytecode is one of the specified target bytecodes, and conditionally execute the instrumentation, but this would add too much overhead.

Instrumentation Overhead. Although our instrumentation could incur noteworthy overhead when enabled on frequently executed bytecode instructions, we only used them to measure a specific bytecode instruction, instead of end-to-end runtime. We only count the number of instructions inside of the measurement instructions, not including our instrumented instructions. This is possible as we control the exact assembly instructions generated, and we verify said assembly using `objdump`, `gdb`, and outputting the JIT-compiled assembly.⁵

However, our cycle measurements could be skewed by the measuring instructions limiting the processor’s out-of-order execution and pipelining capabilities. This is a limitation, and it is extremely difficult to accurately measure this overhead due to the fact that the very act of measuring the cycle count disrupts pipelining (similar to the observer effect in physics).

3 LangBench

Our goal is to compare realistic applications across different languages. We cannot reuse existing benchmarks as they target specific languages, and extending these benchmarks to other languages is infeasible. For example, porting the DaCapo benchmark requires us to implement Eclipse, a full-featured IDE, in four other languages [50]. Therefore, we chose to build 6 applications from the ground up to cover a variety of workloads. We implemented these applications in each of the 5 languages: C++, Go, Java, JavaScript, and Python. From the six applications, we created twelve benchmarks by varying degrees of concurrency, and exploring alternative implementations of the applications.

We made a best-effort attempt at covering a variety of different types of workloads. Our applications range from micro-benchmarks to real world applications, and they stress three major resource usage categories in different ways, being one or more CPU-intensive, memory bound, and I/O bound. Additionally, we implemented parallel versions of the applications where applicable. They also vary from short running to long running ones. The applications and their categories are shown in Table 1.

It was important to ensure that the applications were implemented in a similar and fair manner in each of the five

⁵There is no overhead for removing the type and bounds checking in V8 as the compiler only removes instructions.

Application	CPU	Memory	I/O	Parallel
Sudoku solver	✓			
String sorting	✓			
Graph coloring	✓	✓		
Key-Value store		✓	✓	✓
Log analysis	✓	✓	✓	✓
File server			✓	✓

Table 1: The applications and the component(s) they stress.

languages. The design of every implementation of an application is conceptually identical: they use the same algorithms and control flow. Each application is relatively small, so that it could be built in all languages using the same algorithms and data structures. This kept the complexity of the application similar across the languages.

Yet, we fully rewrote the applications in each language, providing our best effort to make the code idiomatic. We referred to official language documentation (e.g. [7]). In certain cases we also implemented different versions of the code. For example, in the JavaScript Sudoku implementation we re-wrote the code multiple times to change the storage of the arrays (see §6.1 for details). For Python and JavaScript we also tried versions that create parallelism (e.g. multiprocessing in Python) and versions that only provide concurrency (e.g. threading in Python). In general, as we analysed each bottleneck, we also tried to find any more performant implementations.

The six applications are:

Sudoku Solver. We implemented an exhaustive search sudoku solver, borrowing from the Spec CPU 2017 benchmark [30]. The algorithm recursively labels all empty cells. At each cell, it verifies the grid state, using the next digit for the cell if verification fails. If all digits are exhausted for a cell, it backtracks to the previous cell.

String Sorting. We implemented the in-place merge sort algorithm described by Katajainen *et al.* [59] and use it to sort strings. First, we permute every possible string of length 6 with 18 possible letters, creating 18^6 strings. These strings are stored in an array, which are then sorted.

Graph Coloring. Graph coloring labels each vertex in a graph, such that no two vertices with an edge between them have the same label. We implemented the algorithm presented by Wigderson [78] which uses a bounded number of colors with run time complexity polynomial in the number of vertices, edges, and the chromatic number. The benchmark colors the YouTube social network and ground-truth communities graph from the Stanford Large Network Dataset Collection [60]. We implemented both a recursive and an iterative version of the algorithm.

Key-Value Store. We implemented an in-memory key-value store based on the general architecture of Redis [28]. We stress the server with Redis’ packaged benchmark by running a SET test followed by a GET test. Each test makes 2 million requests, randomly selecting a key from a space of 500 thousand keys, using a value size of 64 bytes. The Redis benchmark opens

a configured number of client connections to the key-value store. Each connection performs an equal number of requests and is treated as a unit of concurrency (such as a thread). The clients are run on a machine separate from the one running the key-value store.

Log Analysis. We implemented the algorithm of CLP that parses logs by separating highly repetitive static text from variable values, and stores them in two different indexes [73]. Logs are queried, returning the matching log messages by searching the index and raw log. We separate the searching into two separate tests. “Regex” searches the raw logs using regular expressions, whereas “Indexed” searches using indexes. Both tests can be run with parallelism, where the files to be searched are partitioned equally. We process 7000 log files totalling 1.21 GB on disk with an average size of 181 KB. The logs were generated by running various jobs from HiBench [56, 57]. Each test first indexes the logs and then performs 7 queries.

File Server. We implemented an HTTP server that serves static files from a directory. A single C++ client is always used that spawns a configured number of threads; each thread connects to the server and requests an equal partition of 1000 real log files with an average size of 16.8 MB. The server implementations handle these connections concurrently, treating each connection as a unit of concurrency. The client and server run on different machines.

4 Methodology

We ran our experiments on two in-house servers, each having 2 Xeon E5-2630V3, 16 virtual cores, 2.4 GHz CPUs, 256 GB DDR4 RAM and two 7200 RPM hard drives. They are running Linux 4.15.0 and connected by a 10 Gbps interconnect. For C++ programs we used GCC 9.3.0 compiling with `-O3` against the C++17 standard. For OpenJDK 13 [17], CPython 3.8.1 [25], and Go 1.14.1 [6], we used the reference implementations for each respective language. We use Node.js 13.12.0 [16] which uses V8 7.9.317.25 [41].

We ran each benchmark 5 times and report the average. The key-value store, log parser, and file server benchmarks were run with the number of both client and worker threads ranging from 1 to 1024. For OpenJDK and V8 the minimum amount of memory was set by determining the first heap configuration that did not cause a crash; for Go, GOGC was set to 5%. We then continuously increased the heap settings until performance no longer improved. We used the results from the first setting (i.e., the smallest heap size) that resulted in optimal performance. For the log parser and file server benchmarks, the used log files were stored on a distributed file system with a replication factor of 2. We cleared Linux’s page cache before running each benchmark.

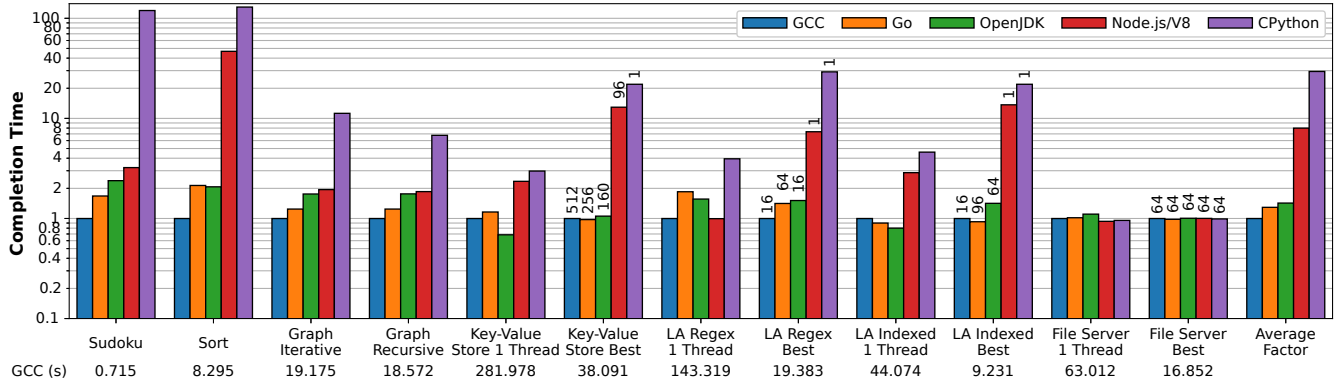


Figure 2: Relative completion times for various language implementations normalized to optimized code under GCC. Note the logarithmic scale of the y axis. “LA” refers to the log analysis application. The numbers at the bottom shows the benchmark’s absolute execution time in the C++ implementation. For benchmarks with concurrency, the “Best” bars are annotated with the thread count that results in best completion time. For key-value store and file server it is the number of client threads, not the number of threads used server side. For GCC and OpenJDK, the server creates 1 (kernel) thread to handle each client thread, so the number of server-side threads is the same as the client. For both Node.js and CPython, their best completion time in key-value store is achieved when using a single server-side thread (due to their scalability characteristic described in §7). As for the file server benchmark, both Node.js and CPython’s best performance is achieved when using 64 server-side threads (§7). The number of server-side threads in Go is automatically determined by the runtime as described in §8.2. The number of threads for log analysis is the number of worker threads (as there is no client).

5 Overview of Results

Figure 2 shows the run times for the benchmarks in LangBench. Unsurprisingly, optimized GCC was the *fastest* on average, with Go and OpenJDK close behind, being 1.30x and 1.43x slower than GCC. Impressively, Go and OpenJDK outperform optimized GCC for 3 out of the 12 benchmarks. V8/Node.js and CPython performed the worst with run times 8.01x and 29.50x slower than GCC. At the extreme, CPython was 129.66x slower than GCC (for the sort benchmark). V8/Node.js and CPython were competitive with GCC only when the workload is bottlenecked by disk I/O, i.e., in the file server benchmark.

We also found that V8/Node.js and CPython are limited with respect to achievable parallelism. Their design serializes the threads’ computation, and requires expensive serialization for different threads (V8) or processes (CPython) to communicate. This leads to the unintuitive result that adding additional threads actually slows down parallel applications as more serialization is required. In fact, for both the key-value store and parallel log analysis benchmark, the best performance is achieved using only a single thread. In contrast, both Go and OpenJDK scale to multicores. Go achieves a 1.02x speedup over GCC in the multithreaded key-value store benchmark, despite being slower in the single threaded version.

In the subsequent sections, we use our instrumented runtimes to provide detailed analyses that explain these results. Specifically, for each runtime, we analyze (at minimum) the two worst performing benchmarks, considering both single-threaded (§6) and multi-threaded (§7) variants for those with parallelism. We further analyzed every case where the runtime outperforms GCC (§8).

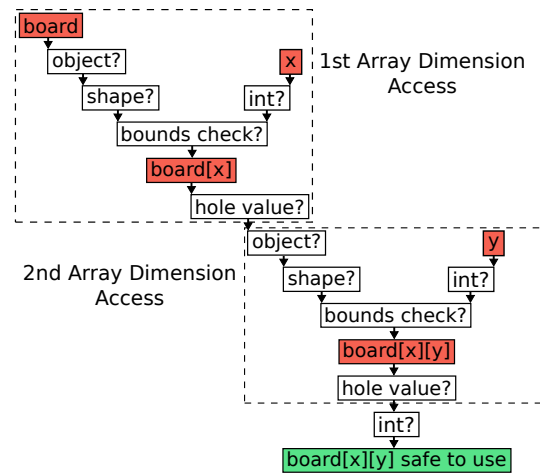


Figure 3: Checks required to access `board[x][y]` in V8/Node.js.

6 Runtime Overhead (Single-thread)

This section investigates the source of runtime overheads on the LangBench single-threaded applications that performed poorly. Specifically, we found (1) type and bounds checking (§6.1) is the bottleneck for V8 in its slowest benchmarks (Sudoku and Sort); (2) interpreter performance (§6.2) is the major cause of CPython’s overhead – despite lacking a JIT compiler, its interpreter performs much worse compared to OpenJDK and V8; (3) GC write barrier (§6.3) is the bottleneck in the Sort benchmark for both OpenJDK and Go, which is their slowest workload, even when heap usage is small.

6.1 Type and Bounds Checking Overhead

We found that type checking and bounds checking made up 41.83% and 87.43% of V8’s execution time in the default su-

Code Version	Time (s)	Overhead of Checks (%)
Default	2.369	–
1-2 Remove Obj./Int Checks	2.177	8.105
3 Remove Shape Check	2.219	6.332
4 Remove Bounds Check	2.154	9.076
5 Remove Hole Check	2.051	13.423
1-5 Remove All Checks	1.378	41.832

Table 2: We modified V8’s JIT compiler and removed each of the checks performed for a 2D array access to `board[x][y]` shown in Figure 3. We measured the resulting execution time, and compare it against the default execution time with all checks. We also show the execution time when all checks are removed.

doku and sort benchmarks, which are the two single-threaded benchmarks where V8 showed the worst performance compared to GCC. Note that V8 has other sources of overhead, such as execution being partially interpreted, when compared with GCC. For the numbers in this sub-section, we compare against the default execution time when the runtime binary is in Linux’s page cache, unlike the results in Figure 2, where we clear the page cache before each test. Next we zoom into the Sudoku benchmark to explain this overhead, and how we can leverage our bytecode profiling information to optimize our JavaScript code.

For V8/Node.js, Sudoku spends 93% of its time primarily comparing 2D array elements of the sudoku board. The majority of this time is spent performing 11 type and bound checks for each 2D array access, as shown in Figure 3. Each dimension requires 5 checks, and the 11th check is used for the final value. Table 2 shows the overhead for these checks.

The first check ensures that `board` is an object pointer, by checking for a tagged bit to distinguish between an object pointer and a primitive integer value. (V8 stores both object pointers and primitive integers in a 8 byte word, so that integers can be stored inline instead of being allocated on the heap.) Second, V8 must similarly check that `x` is an integer, rather than an object. Omitting these checks made it 8.1% faster (shown in Table 2) — removing them is safe, as we know that no incorrect type will be used.

After V8 confirms that `board` is an object, it checks that the internal type of `board`, called a shape, is an array. Fourth, V8 performs a bounds check for the access to `board[x]`. Finally, V8 checks if the value accessed is a “hole”. In JavaScript, arrays may be sparse, meaning not every index has a value. Indexes without values are called holes, which V8 must convert to `undefined` if accessed. The same checks must be repeated to access the second dimension of `board`. To use `board[x][y]`, a last check is necessary to verify it is an integer.

Profiling enabled optimization. Initially, we preallocated the fix-size sudoku board. In V8, preallocated arrays are created sparse as their values are uninitialized, requiring the hole checks. Even though the array was filled with integers before being used, sparse arrays never lose their status.

```
// OpenJDK: bounds check board[x].length > 8
for (int i = 0; i < 9; i++) {
    // Go: bounds check board[x].length > i
    int element = board[x][i];
    ...
}
```

Figure 4: Code showing where Go and OpenJDK perform array bounds checking when accessing `board[x][i]` in a loop.

	Bytecode	Insn. per BC	Cycles per BC
OpenJDK	Assembly aaload	12	7.7
	Assembly iaload	11	7.1
	C++ aaload	33	12.5
	C++ iaload	22	11.1
Node.js	LdaKeyedProperty	90	26.3
CPython	BINARY_SUBSCR	138	41.8

Table 3: Statistics for array access bytecodes (BC) performed by various interpreters for the sudoku benchmark.

We implemented an optimized version which would create arrays without holes, known as “packed” arrays. This optimized version was 1.48x faster (and is what is shown in Fig. 2). Our optimized sudoku benchmark for V8/Node.js starts with an empty array, then appends 9 `Int8Arrays` to create the 2D sudoku board. This allows V8 to recognize that there are no holes. Using the built in `Int8Array`, preallocation initialized it with the default value of 0, rather than a hole.

Unfortunately, these optimizations cannot be applied universally. First, it presents a trade-off that can only be determined via profiling: while sparse arrays require hole checking, building a large packed array requires many internal resizing operations to grow the array. In addition, typed arrays such as `Int8Array` only exist for certain integral types. For example, it is not possible to preallocate a packed array of strings or any user defined type.

GCC, Go, and OpenJDK. Compared to GCC, Go and OpenJDK must also perform similar bounds checks. However, they avoid the type checking as they are statically typed.

OpenJDK successfully lifts all loop-invariant computations to outside the loop. In the code of Figure 4, OpenJDK determines that the maximum value of `i` used to access `board[x][i]` is 8, and checks if the length of `board[x]` is greater than 8 outside the loop. Go performs the bounds check in each iteration. Further, 2D arrays in OpenJDK contain pointers to 1D arrays, which may be null. However, OpenJDK has an optimization which eliminates null checks, and instead catches them using a signal handler for `SIGSEGV`. On the other hand, Go does not need to perform null checks as its 2D arrays are laid out contiguously in memory like in C.

6.2 Interpreter Overhead

CPython is slower than the other runtimes because it lacks a JIT compiler and so programs are strictly interpreted. There-

fore we further compare the three runtimes by running sudoku on each of them only in interpreter mode. OpenJDK's (assembly) interpreter outperforms both V8 and CPython by 2.59x and 5.34x respectively. This is because static typing allows OpenJDK to avoid the type checks that V8 and CPython must perform. OpenJDK has dedicated bytecodes for accessing different types of arrays (`aaload` for an array of arrays, `iaload` for an integer array). In contrast, V8 and CPython both have a single bytecode (`LdaKeyedProperty` and `BINARY_SUBSCR`, respectively) which must accommodate for any array or dictionary type. Table 3 shows the performance profiling results of different bytecode executions, using our instrumentations.

CPython is still 2.07x slower than V8, even though both of them do dynamic typechecking. As shown in Table 3, CPython uses 138 instructions and 41.8 cycles to execute each byte code instruction (`BINARY_SUBSCR`), whereas Node.js only spends 90 instructions/26.3 cycles to process each byte code instruction (`LdaKeyedProperty`). This is due to the optimizations of V8's interpreter: it is hand-crafted in IR, whereas CPython is implemented in C. Similarly, we found that OpenJDK's assembly interpreter is 1.93x faster than the one implemented in C++.

We found the hand-crafted interpreter implementation made a few notable optimizations. First, it aggressively inlined functions. The CPython bytecode we inspected ended up containing around 5-6 function calls in the common execution path. The equivalent bytecode in Node.js had no call instructions, similar to when all functions are completely inlined. This further enables more aggressive optimization. For example, error handling logic that would be functions in C code can now be grouped together at the end. This leaves the instructions in the non-error paths tightly together and improves cache performance. In addition, developers have a better understanding than the compiler on what the common path is, so that they can manually group the basic blocks that are commonly executed together (and move error handling logic to the end).

Theoretically GCC could also perform the same level of inlining, and developers can manually use `goto` statements to move all error handling logic to after the common-path logic. However, performing aggressive inlining unselectively could hurt performance (increased function size, register pressures, etc.), and excessive use of `goto` could hurt the readability, reliability, and maintainability of the software.

Performance Sensitivity on Interpreters. We observe that an interpreter may amplify the performance overhead caused by small code changes that would only incur negligible overhead in compiled execution. Under CPython, the iterative version of graph coloring ran 1.66x slower than the recursive version, in contrast to the other interpreters. The function performing the iterative algorithm contained 1.54x more bytecodes than the recursive function, resulting in 22% more instructions recorded by `perf`. In contrast, for GCC, switching from recursive to iterative adds only 4% more instructions.

Iterative versions of recursive functions are commonly necessary to avoid stack overflows. Instead of a recursive function call, the iterative function appends the call arguments to an array, and later pops the arguments off the array to perform another iteration of the algorithm. In addition, the iterative algorithm must check if the array is empty at each iteration of the loop. These seemingly simple operations significantly increase the bytecode count and execution time. JIT compilers mitigate these extra operations through optimizations such as reducing the number of redundant checks.

Startup Overhead for OpenJDK and V8. OpenJDK and V8 spend 843ms and 788ms, respectively, in startup in the Sudoku benchmark. Startup is the primary reason for OpenJDK being slower than GCC when running Sudoku, as it is the shortest benchmark. Specifically, 708ms is spent loading the JVM's large binary from disk, while the rest (135ms) is spent in classloading and interpreter execution. In comparison, OpenJDK's warm execution time is only 868ms (when we run the Sudoku benchmark in a JVM that has already been warmed up by running the same benchmark multiple times), whereas GCC's warm execution time is 611ms.

6.3 GC Write Barriers

We were surprised to see that under OpenJDK's default GC setting, it was 10.03x slower than GCC for the Sort benchmark. Sort is also the benchmark where Go performs the worst relative to GCC: 2.14x slower. The source of the slowdown for both OpenJDK and Go is the cost of GC write barriers. This cost occurs despite GC hardly ever running in Sort, as write barriers are necessary to maintain data structures needed to perform GC. Interestingly, for OpenJDK, using the non-default Parallel GC algorithm drops the slowdown to only 2.07x (shown in Figure 2), as it contains fewer instructions. Go's write barrier contains even fewer instructions, and is slightly faster than OpenJDK with Parallel GC.

Write barriers bring a constant cost to pointer writes regardless of how often GC is actually performed. For our in-place merge sort, swapping two elements is the primary source of write barriers. This requires two write barriers, one for each element being written. OpenJDK's default GC algorithm, G1 [12], adds 44 instructions for these write barriers, completely dwarfing the 6 instructions required to swap the elements and 5 for bounds checking. On the other hand, Parallel GC's write barriers only use 5 instructions, 8.8x fewer than G1.

Both Go and G1 require a write barrier to ensure every live object is captured when they perform marking concurrently with application threads. Furthermore, both G1 and Parallel GC in OpenJDK divide the heap into regions and move live objects across regions to compact the heap. For both, write barriers are used to maintain remembered sets, which are used to find and update pointers to moved objects. However,

G1 performs more checks during its write barrier to avoid unnecessary updates to the remembered sets. This avoids work when using the remembered sets to update pointers, and helps minimize pause time.

7 Scalability Limitations

We found that CPython and Node.js limit the degree of parallelism achievable. We first briefly introduce the background of the Node.js and CPython concurrency model and then describe our findings.

Background. Node.js is event driven; by default, it uses a single Node.js thread to drive an event loop and process all incoming events. If the processing of an event blocks (e.g., on I/O), the underlying kernel thread will block, and Node.js's event loop continues with another kernel thread to process the next event. In other words, multiple threads can be blocked at the same time, but CPU execution is serialized. While Node.js supports running multiple Node.js threads (known as worker threads), each runs its own event loop. Hence worker threads *do not* share the heap (to avoid data races); data sharing requires message passing with data being serialized.

In essence, CPython's concurrency model is the same as that of Node.js where multiple kernel threads can block on I/O at the same time, except that it is the programmer's job to create the threads; the threads share the same heap. In CPython's case, CPU computation is serialized by the Global Interpreter Lock (GIL) so that only one thread can use the CPU at a time. CPython also supports `multiprocessing`, forking different processes to avoid the GIL. However, data sharing and communication requires serialization.

Node.js and CPython's scalability on LangBench. We can now explain the scalability patterns of Node.js and CPython. We ran three parallel benchmarks, namely log analysis, key-value store, and file server, under different configurations, including different number of threads, as well as parallelizing them with multiple processes in CPython. In log analysis and key-value store, *the best performance is achieved using a single CPython or Node.js thread*, whereas the other runtimes are able to improve performance by adding more threads.

These two benchmarks, namely log analysis and key-value store, are bottlenecked by CPU or memory accesses, instead of blocking I/O. Therefore, creating multiple threads offers no advantage in Node.js and CPython as their executions are serialized. In the case of Node.js, performance degrades significantly when creating additional worker threads due to the serialization overhead. On indexed search log analysis, Node.js's performance drops 4.7x when we use more than one worker thread. In this benchmark, multiple workers communicate frequently as they share the same dictionaries. Similarly, serialization overhead slows down CPython when we switch to `multiprocess`, resulting in a 4.9x slowdown on the same benchmark. While multiple CPython threads share the heap,

they still introduce thread management overhead compared to using a single thread.

Specifically, in key-value store, CPython can only scale to one client thread (adding additional concurrent client threads will worsen the completion time). In comparison, Node.js/V8 scales up to 96 client threads, even though it only uses 1 Node.js event-loop thread at server side. However, its completion time can not keep improving with more client threads, whereas it still can under GCC, Go, and OpenJDK. Note that the improvement plateaus when the client thread count increases to 160. Even though GCC achieved its best completion time on 512 client threads, the improvement over 160 threads is negligible. This is why in Figure 2, the difference in best completion times is small between GCC, Go, and OpenJDK, even though they are achieved on 512, 256, and 160 client threads, respectively.

In comparison, Node.js and CPython scale well on the file server benchmark. This benchmark is I/O parallel: there is little communication between different threads, and they are bottlenecked by disk I/O. Creating multiple threads (or processes in CPython) thus improves the performance (when there are concurrent client connections).

8 Runtime Advantages

We found that the high-level abstractions provided by the runtimes can, in some cases, result in better performance and scalability. This is counter-intuitive given the conventional wisdom that abstractions generally come at the expense of performance [74]. We discuss three findings: (1) object relocations in OpenJDK's moving GC can result in better cache locality; (2) Go's scheduler automatically maps user threads to kernel threads, and hence abstracts away the direct usage of kernel threads, reducing the number of context switches and the number of kernel threads used; (3) abstracting away the low-level I/O operations allows runtimes to use the optimal I/O system call configurations.⁶

8.1 GC Improved Cache Locality

OpenJDK's moving garbage collector can significantly improve cache locality, resulting in speedups in three benchmarks: single threaded key-value store and both iterative and recursive implementations graph coloring. In particular, OpenJDK was much faster than GCC at the single threaded key-value store, with 1.46x speedup. This is the largest speedup any runtime had over GCC.

Key-value store. We found that the source of cache locality was from iterating over linked lists. Our key-value store implements a hashtable with separate chaining, meaning hash

⁶There are a few more cases where runtimes demonstrated better performance than GCC; they are related to the implementation of libraries. We discuss them in detail in the Appendix.

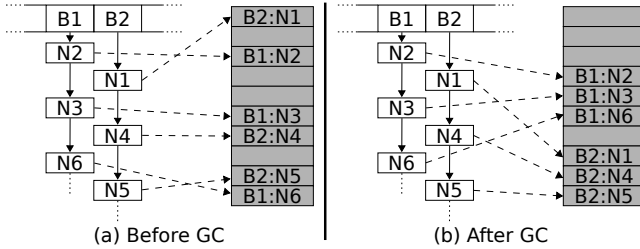


Figure 5: The key-value store before and after a GC pause. White boxes logically represent Java objects, and the shaded boxes represent the objects’ location in the JVM heap. A ‘B’ denotes a bucket mapped to by the hash function, and an ‘N’ denotes a node in the bucket’s linked list. The number of the node represents the order they are inserted into the hashtable. The memory for the nodes of the bucket begins scattered, but after GC relocation is ordered by the traversal of the bucket’s linked lists.

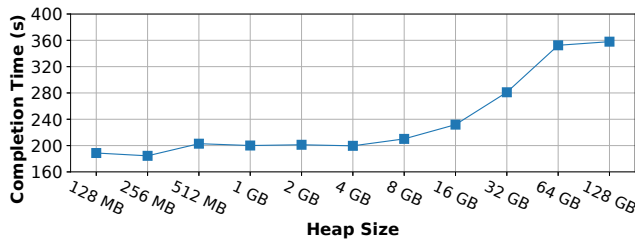


Figure 6: The OpenJDK single threaded key-value store benchmark run with increasing heap sizes, corresponding to fewer GC cycles.

collisions are added to a bucket by appending the key-value (KV) pair to a list. This is shown as the white boxes in Figure 5. For example, N2, N3, and N6 are different KV pairs hashed to the same bucket B1.

OpenJDK uses bump pointer allocation. Therefore, nodes in the hashtable are laid out sequentially in memory based on their insertion order. Figure 5 (a) shows how the nodes of a bucket would be initially laid out in memory. There is little locality, as adjacent nodes of the same linked list are scattered. Therefore, whenever there is a lookup, insertion, or a deletion of a key in the linked list, the traversal of the linked list is expensive due to poor locality.

However, OpenJDK’s moving GC reorders the objects in memory. It scans for all live objects that are reachable from the GC roots (e.g., objects on the stack) by following the pointers, copying them to a different memory region, before freeing the old region. For the linked list, this means that the objects will be allocated adjacently, in the same order as in the linked list, as shown in Figure 5 (b).

In comparison, GCC uses a size segregated allocator (`malloc`). Since nodes have the same size, they will be placed in the same region, resulting in a similar pattern as with bump pointer allocation, with nodes laid out in insertion order. When profiling the iteration, we found that GCC actually executed fewer instructions than OpenJDK, but was still slower. In the tight loop iteration, the bucket GCC took only 5 assembly instructions compared to OpenJDK’s 11.

This behavior presents the unintuitive case where the more

frequently GC is performed, the better the performance. Figure 6 shows that with more frequent GC cycles, objects are re-ordered in memory more often, leading to improved performance. We control the frequency of GC by using different heap sizes. The larger the heap, the fewer GC cycles. When it is 128 GB, performance is the *worst* because GC is never triggered; objects are never moved, so there is no locality.

To verify that cache locality was the source of the performance gap, we modified OpenJDK to expose a method to print the virtual address a reference points to. We do this as GC obfuscates `perf` cache hit rates, making them impractical to compare. In one run where no GC was performed, over 99% of the distances between nodes of the linked lists were different, with a median distance of 724 KB. A run with GC was 1.86x faster; 57% of nodes were 88 bytes apart, and 41% were 192 bytes apart. Although the size of a cache line on our processors is 64 bytes, so two nodes would not be in the same cache line, it is likely that they are in adjacent cache lines, opening the opportunity for prefetching.

Graph coloring. We found OpenJDK outperformed GCC (by 1.37x) on graph coloring, when the C++ program uses the standard library. Our investigation showed that GC had a similar effect as for the key-value benchmark given that graph coloring also uses a hash table. Both hash table implementations on OpenJDK (`HashMap` and `HashSet`) and C++’s standard libraries (`std::unordered_set` and `std::unordered_map`) use an open hashing design; i.e., it uses separate chaining to connect the elements in a linked list upon collision. As a result, both GCC and OpenJDK suffer from poor locality initially. However, OpenJDK quickly gains locality through GC, as with the key-value store benchmark.⁷

8.2 Scalability in Go

In the multithreaded key-value store implementation, Go has a 1.02x speedup compared to GCC, despite being 1.16x slower than GCC in the single threaded version. Go outperforms GCC by avoiding 2.2 million context switches through the use of asynchronous networking I/O and significantly fewer kernel threads. With GCC, network I/O is performed using synchronous system calls, blocking the kernel thread, resulting in a context switch. When goroutines perform I/O, the work is offloaded to an internal goroutine which uses asynchronous system calls. A goroutine performing I/O is blocked by Go’s scheduler, but the underlying kernel thread is not blocked; instead, Go schedules another goroutine on the same kernel thread. As a result, Go only uses at most 42 kernel threads, regardless of the number of concurrent client threads. (The number of kernel threads is automatically chosen by the Go runtime depending on the workload’s characteristic.)

⁷We optimized our C++ benchmark by switching to hashtable implementations from Google’s `Abseil` library [2], which uses a closed hashing implementation that achieves better locality.

We verified that context switching causes the majority of the 600 ms gap between the fastest multithreaded Go and GCC execution times. Using LEBench [72], we measured the average cost of a context switch on our machine to be 5.84 μ s. With 32 cores, `perf` reports approximately 70K context switches per core, which adds up to 409ms of overhead, making up the majority of the 600ms performance gap.

8.3 I/O System Calls in the File Server

To read a file in the file server benchmark in C++, we initially used the more general, idiomatic approach which uses iterators. This results in repeated fixed size `read` system calls. Unlike C++, all the managed runtimes abstract away the low-level system call interfaces when performing I/O, so that they can transparently issue system calls in an optimal way, by first calling `fstat` to get the file size, followed by *a single read* for its entire contents. All runtimes use this approach when reading a file. So any developer using the runtimes will benefit from the optimizations without any burden of knowledge. In comparison, we have to manually optimize our C++ implementation to switch to `fstat` and `read`, leading to a 2x speedup.

9 Related Work

Ours is the first performance study to analyze and compare the implementations of multiple widely used runtimes, and provide the necessary instrumentations to do so. There are existing benchmarks to evaluate software performance, but they focus on novel benchmark methodologies. Marr *et al.* designed a benchmark suite with the goal of having a methodology for evenly comparing a common subset of language abstractions [64]. They limit their applications to a minimal set of primitive operations and exclude built-in data structures such as hashables to ensure that no language has an advantage. Rather than strictly stressing the compiler on specific primitive operations, we evaluated all aspects of a runtime on how they affect performance under different scenarios using idiomatic code. Both DaCapo [50] and Renaissance [70] created benchmark suites consisting only of Java applications for various workloads. TailBench created a statistically sound methodology for measuring latency-critical applications in C++ and Java [58]. SPEC [31] and the Computer Language Benchmarks Game [3] provide a variety of benchmarks, covering many languages, but present no analysis. In contrast, our work focuses on understanding and providing an explanation for the technical details of language implementations that cause performance differences.

Other studies of languages have had different scopes, focusing tightly on a specific language or aspect. By utilizing the Rosetta Code [29] repository, Nanz *et al.* present statistical findings, such as the fact that scripting languages are more concise than procedural languages [67]. Nanz *et al.* further

studied the usability and performance of Chapel, Cilk, and Go in multicore workloads [66]. Prokopski *et al.* study interpreter code-copying optimizations in the SableVM, OCaml, and Yarr interpreters [71]. Wade *et al.* quantify the impact of profile data on JIT compiled code quality in the HotSpot VM.

Lion *et al.* instrumented the JVM to measure startup times (i.e., the total time spent in class loading and interpreter) [62]. However, they did not provide fine-grained instrumentation to profile the execution of each bytecode instruction.

10 Concluding Remarks

We presented an in-depth performance analysis of runtimes under a variety of scenarios. We implemented LangBench, a benchmark suite that enables an objective comparison of language implementation performance. Our runtime instrumentations facilitate understanding *why* a runtime performs well or poorly. We demonstrated that our instrumentations provide valuable profiling information that enables optimizations. We have open-sourced our instrumentations and LangBench so that practitioners can use and enhance them to analyze and optimize their applications.

Acknowledgements

We thank the anonymous reviewers and the shepherd for their insightful comments. This research was supported by the Canada Research Chair fund, an NSERC Discovery grant, and a VMware gift.

References

- [1] 10 Myths of Enterprise Python. <https://medium.com/paypal-engineering/10-myths-of-enterprise-python-8302b8f21f82>.
- [2] Abseil. <https://abseil.io/>.
- [3] The Computer Language Benchmarks Game. <https://benchmarksgame-team.pages.debian.net/benchmarksgame/>.
- [4] etcd - A distributed, reliable key-value store for the most critical data of a distributed system. <https://etcd.io/>.
- [5] Go memory ballast: How I learnt to stop worrying and love the heap. <https://blog.twitch.tv/en/2019/04/10/go-memory-ballast-how-i-learnt-to-stop-worrying-and-love-the-heap-26c2462549a2/>.
- [6] The Go Programming Language. <https://golang.org/>.

- [7] Go Programming Language Documentation. <https://go.dev/doc/>.
- [8] Go Programming Language Specification. <https://golang.org/ref/spec>.
- [9] Introduction to Intel Advanced Vector Extensions. <https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html>.
- [10] Java EE: DayTrader Benchmark. <https://github.com/OpenLiberty/sample.daytrader8>.
- [11] JavaScript is slow. <https://kariera.future-processing.pl/blog/javascript-is-slow/>.
- [12] JEP 248: Make G1 the Default Garbage Collector. <http://openjdk.java.net/jeps/248>.
- [13] Linux source tsc_sync.c: Check tsc synchronization. https://github.com/torvalds/linux/blob/df0cc57e057f18e44dac8e6c18aba47ab53202f9/arch/x86/kernel/tsc_sync.c.
- [14] M3: Uber's Open Source, Large-scale Metrics Platform for Prometheus. <https://eng.uber.com/m3/>.
- [15] Most popular languages on GitHub. <https://github.com/oprogramador/github-languages>.
- [16] Node.js. <https://nodejs.org/en/>.
- [17] OpenJDK 13. <https://openjdk.java.net/projects/jdk/13/>.
- [18] OpenStack Overview. <https://www.openstack.org/software/>.
- [19] Optimizing a Golang service to reduce over 40% CPU. <https://cotalogix.com/log-analytics-blog/optimizing-a-golang-service-to-reduce-over-40-cpu/>.
- [20] Our journey to type checking 4 million lines of Python. <https://blogs.dropbox.com/tech/2019/09/our-journey-to-type-checking-4-million-lines-of-python/>.
- [21] Production-Grade Container Orchestration - Kubernetes. <https://kubernetes.io/>.
- [22] Profiling CPython at Instagram. <https://instagram-engineering.com/profiling-cpython-at-instagram-89d4cbeeb898>.
- [23] Project Tungsten: Bringing Apache Spark Closer to Bare Metal. <https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html>.
- [24] PYPL PopularitY of Programming Language. <http://pypl.github.io/PYPL.html>.
- [25] Python Implementations - Python Wiki. <https://wiki.python.org/moin/PythonImplementations>.
- [26] Quora: In what cases is Java faster than C. <https://www.quora.com/In-what-cases-is-Java-faster-if-at-all-than-C>.
- [27] Quora: In what cases is Java slower than C by a big margin. <https://www.quora.com/In-what-cases-is-Java-slower-than-C-by-a-big-margin>.
- [28] Redis. <https://redis.io>.
- [29] Rosetta Code. https://rosettacode.org/wiki/Rosetta_Code.
- [30] SPEC CPU 2017 Documentation. <https://www.spec.org/cpu2017/Docs/#benchmarks>.
- [31] SPEC: Standard Performance Evaluation Corporation. <https://www.spec.org>.
- [32] SPECjbb 2015 Benchmark. <https://www.spec.org/jbb2015/>.
- [33] Stack Overflow: C++11 regex slower than python. <https://stackoverflow.com/questions/14205096/c11-regex-slower-than-python>.
- [34] Stack Overflow: Why do std::string operations perform poorly? <https://stackoverflow.com/questions/8310039/why-do-stdstring-operations-perform-poorly>.
- [35] Stack Overflow: Why is python faster than c++ in this case? <https://stackoverflow.com/questions/24895881/why-is-python-faster-than-c-in-this-case>.
- [36] The State of Developer Ecosystem 2019. <https://www.jetbrains.com/lp/devecosystem-2019/>.
- [37] The State of Serverless. <https://www.datadoghq.com/state-of-serverless/>.
- [38] The State of the Octoverse. <https://octoverse.github.com>.
- [39] Transducers Speed Up JavaScript Arrays. <https://itnext.io/using-transducers-to-speed-up-javascript-arrays-92677d000096>.
- [40] Twitter Shifting More Code to JVM, Citing Performance and Encapsulation As Primary Drivers. <https://www.infoq.com/articles/twitter-java-use/>.
- [41] V8 JavaScript engine. <https://v8.dev/>.

- [42] Why did Twitter switch from Ruby on Rails? <https://medium.com/@mittalyashu/why-did-twitter-switch-from-ruby-on-rails-dac66150044d>.
- [43] Why Discord is switching from Go to Rust. <https://blog.discord.com/why-discord-is-switching-from-go-to-rust-a190bbca2blf>.
- [44] Why is Dynamic Type Checking Expensive? <https://stackoverflow.com/questions/41622341/why-is-type-checking-expensive>.
- [45] Why the Hell Would You Use Node.js. <https://medium.com/the-node-js-collection/why-the-hell-would-you-use-node-js-4b053b94ab8e>.
- [46] Why we switched from Python to Go. <https://getstream.io/blog/switched-python-go/#reason-performance>.
- [47] Yes, Python is Slow, and I Don't Care. <https://medium.com/pyslackers/yes-python-is-slow-and-i-dont-care-13763980b5a1>.
- [48] Intel® 64 and IA-32 architectures software developer's manual, Volume 3B: System programming guide, part 2, Section 17.15. <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf>, 2016.
- [49] Jeffrey Barber, Ximing Yu, Laney Kuenzel Zamore, Jerry Lin, Vahid Jazayeri, Shie Erlich, Tony Savor, and Michael Stumm. Bladerunner: Stream processing at scale for a live view of backend data mutations at the edge. In *Proc. 28th ACM Symp. on Operating Principles (SOSP'21)*, page 708–723. Association for Computing Machinery, October 2021.
- [50] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proc. 21st Conf. on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, pages 169–190. ACM, 2006.
- [51] Rodrigo Bruno, Paulo Ferreira, Ruslan Synytsky, Tetiana Fydorenchuk, Jia Rao, Hang Huang, and Song Wu. Dynamic vertical memory scalability for OpenJDK cloud applications. In *Proc. Intl. Symp. on Memory Management (ISMM'18)*, pages 59–70. ACM, 2018.
- [52] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *Proc. 13th Symp. on Operating Systems Design and Implementation (OSDI'18)*, pages 89–105. USENIX Association, October 2018.
- [53] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proc. 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*, page 57–76. Association for Computing Machinery, 2007.
- [54] Hadoop. <https://hadoop.apache.org>.
- [55] Handra. Comparing Hotspot and OpenJ9. <https://www.linkedin.com/pulse/comparing-hotspot-openj9-handra-/>.
- [56] Shengsheng Huang, Jie Huang, Jinqun Dai, Tao Xie, and Bo Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *New Frontiers in Information and Software as Services*, pages 209–228. Springer, 2011.
- [57] Shengsheng Huang, Jie Huang, Yan Liu, Lan Yi, and Jinqun Dai. HiBench: A representative and comprehensive Hadoop benchmark suite. In *Proc. ICDE Workshops, ICDEW '16*. IEEE Press, 2010.
- [58] H. Kasture and D. Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *Proc. IEEE Intl. Symp. on Workload Characterization (IISWC'16)*, pages 1–10. IEEE Press, Sep. 2016.
- [59] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. Practical in-place mergesort. *Nordic J. of Computing*, 3(1):27–40, March 1996.
- [60] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. <http://snap.stanford.edu/data>, June 2014.
- [61] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. The Java® Virtual Machine Specification - Java SE 13 Edition. <https://docs.oracle.com/javase/specs/jvms/se13/html/>.
- [62] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grevski, and Ding Yuan. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. In *Proc. 12th Symp. on Operating Systems Design and Implementation (OSDI'16)*, pages 383–400. USENIX Association, November 2016.

- [63] David Lion, Adrian Chiu, and Ding Yuan. M3: End-to-end memory management in elastic system software stacks. In *Proc. 16th European Conf. on Computer Systems (EUROSYS'21)*, page 507–522. Association for Computing Machinery, 2021.
- [64] Stefan Marr, Benoit Dalozé, and Hanspeter Mössenböck. Cross-language compiler benchmarking: Are we fast yet? In *Proc. 12th Symp. on Dynamic Languages (DLS'16)*, pages 120–131. Association for Computing Machinery, 2016.
- [65] Colt McAnlis. Improving cloud function cold start time, Google Cloud Performance Atlas. <https://medium.com/@duhroach/improving-cloud-function-cold-start-time-2eb6f5700f6>.
- [66] S. Nanz, S. West, K. S. d. Silveira, and B. Meyer. Benchmarking usability and performance of multicore languages. In *Proc. Intl. Symp. on Empirical Software Engineering and Measurement*, pages 183–192. IEEE Press, 2013.
- [67] Sebastian Nanz and Carlo A. Furia. A comparative study of programming languages in Rosetta code. In *Proc. 37th Intl. Conf. on Software Engineering (ICSE'15)*, page 778–788. IEEE Press, 2015.
- [68] Gabriele Paoloni. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. Intel Coporation, 2010.
- [69] Marius Pirvu. Optimize JVM start-up with Eclipse OpenJ9. <https://developer.ibm.com/articles/optimize-jvm-startup-with-eclipse-openj9/>.
- [70] Aleksandar Prokopec, Andrea Rosà, David Leopoldseeder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: Benchmarking suite for parallel applications on the JVM. In *Proc. 40th Conf. on Programming Language Design and Implementation (PLDI'19)*, pages 31–47. Association for Computing Machinery, 2019.
- [71] Gregory B. Prokopski and Clark Verbrugge. Analyzing the performance of code-copying virtual machines. In *Proc. 23rd Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA'08)*, page 403–422. Association for Computing Machinery, 2008.
- [72] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of Linux's core operations. In *Proc. 27th ACM Symp. on Operating Systems Principles (SOSP'19)*, page 554–569. Association for Computing Machinery, 2019.
- [73] Kirk Rodrigues, Yu Luo, and Ding Yuan. CLP: Efficient and scalable search on compressed text logs. In *Proc. 15th Symp. on Operating Systems Design and Implementation (OSDI'21)*, pages 183–198. USENIX Association, July 2021.
- [74] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4):277–288, November 1984.
- [75] Hang Shao, Marius Pirvu, Tobi Ajila, and Vijay Sundaresan. Innovations for Java running in containers. <https://blog.openj9.org/2021/06/15/innovations-for-java-running-in-containers/>.
- [76] Spark. <http://spark.apache.org>.
- [77] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. In *Proc. 14th European Conf. on Computer Systems (EUROSYS'19)*. Association for Computing Machinery, 2019.
- [78] Avi Wigderson. Improving the performance guarantee for approximate graph coloring. *J. ACM*, 30(4):729–735, October 1983.
- [79] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Proc. 7th Symp. on Operating Systems Design and Implementation (OSDI'06)*, pages 103–116. USENIX Association, 2006.
- [80] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with Serverlessbench. In *Proc. 11th ACM Symp. on Cloud Computing (SOCC'20)*, page 30–44. Association for Computing Machinery, 2020.

Appendix

We discuss two additional results in the Appendix: (1) memory usage analysis of different runtimes on LangBench, and (2) other speedups from the runtimes over GCC.

Resource Usage: Memory

Figure 7 shows the peak memory usage of the different runtimes. Compared to Figure 2, it also shows the completion time under the minimum memory usage configuration (e.g., the heap size setting in OpenJDK) of each benchmark. Recall that, for OpenJDK and V8, the minimum amount of memory was set by determining the first heap configuration that did not cause a crash; for Go, GOGC was set to 5%. We then

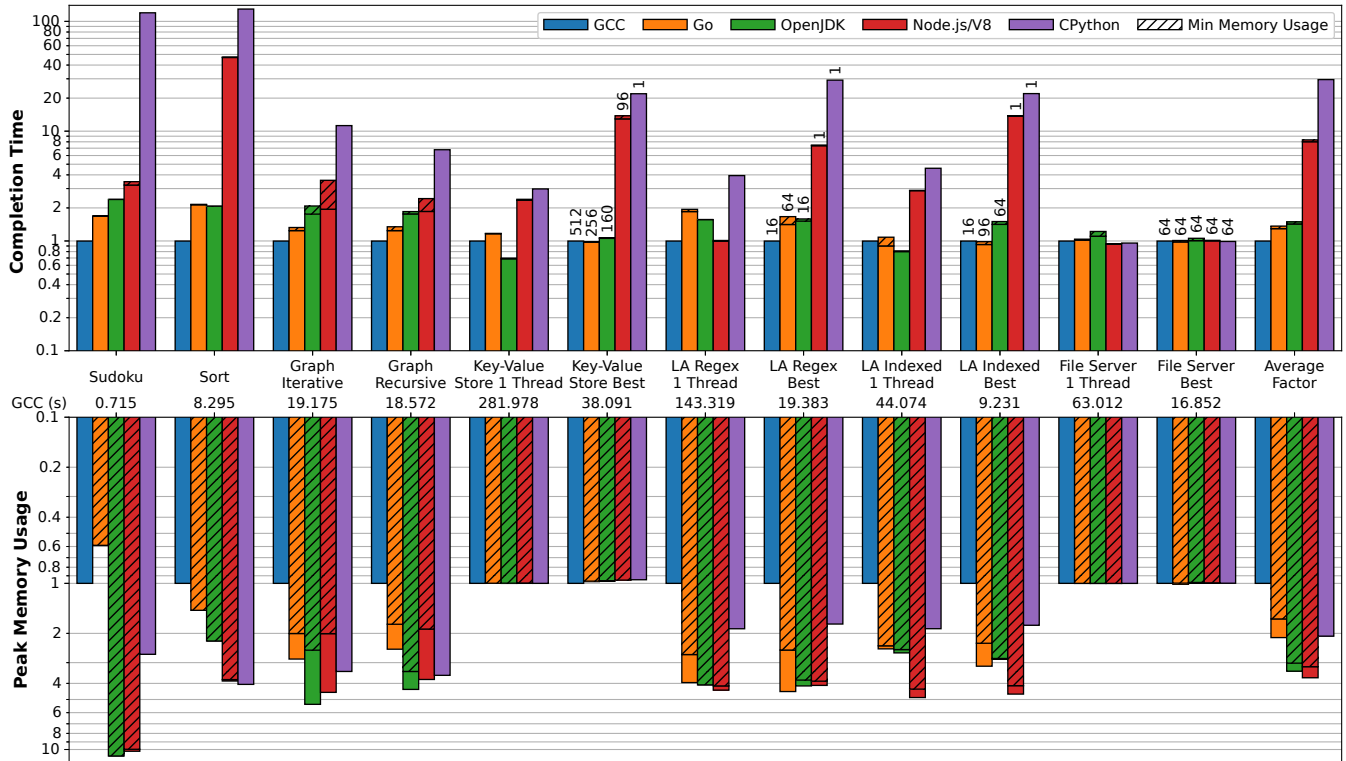


Figure 7: Relative completion time and peak memory usage for various language implementations as a multiplicative factor compared to optimized code under GCC. Each benchmark uses the most optimized version for that language implementation.

continuously increased the heap settings until performance no longer improved. Peak memory was measured using the reported maximum RSS from `getrusage`.

The figure shows that all language implementations use at least 2x more memory than GCC. The more complex runtimes are the worst offenders with V8/Node.js and OpenJDK using 3.70x and 3.38x more memory than GCC on average. Go and CPython use less memory, but still use 2.12x and 2.08x more memory than GCC on average. (As an exception, Go surprisingly manages to use 0.59x less memory than our idiomatic C++ version of the sudoku benchmark.) It is crucial for these runtimes to trade off increased memory usage and performance. Optimal performance can require increased memory usage, which prevents jobs from being scheduled when datacenters allocate resources to fit peak usage. This additional memory is also rarely returned to the OS causing reduced memory utilization.

For both OpenJDK and V8/Node.js, the two runtimes that require the most memory to achieve optimal performance, their worst case was the sudoku benchmark with OpenJDK using 10.94x more memory than GCC. The sort benchmark has the lowest memory usage with GCC only requiring 3.42MB. However, OpenJDK and V8/Node.js also had benchmarks that did not have any memory overhead when compared to GCC. Both runtimes used the same amount of memory as GCC for the key-value store benchmark, despite it being the next smallest benchmark with GCC requiring only 33.43MB.

CPython’s peak memory usage was the closest to that of GCC, but requiring 2.12x more memory on average. It also had the lowest worst case, requiring 4.06x more than GCC for the sort benchmark. Despite being more memory efficient than the other runtimes in most of the benchmarks, CPython still used more memory than any other runtimes in the sort benchmark. Go was also able to use less memory than CPython for the sudoku and graph colouring benchmarks.

In fact, Go was the only language implementation able to use noticeably less memory than GCC, using 0.59x less in the sudoku benchmark. Upon inspection this stemmed from our version of the benchmark using the C++ standard library. The complete C++ version peaks at 3.46MB of RSS, but almost all of this memory is allocated immediately upon running the program. We found that using a C++ implementation that did not use `iostreams` but instead used `open` and `read` reduced the memory usage to 2.72MB. However, the largest improvement was from not linking the C++ standard library by removing the `-lstdc++` flag when compiling. This dropped the usage to 1.33MB and well under Go’s 2.05MB.

Runtime Speedup from Library Implementations

In addition to the cases discussed in §8, there are other cases where managed runtimes performed better than GCC. Go performed better than GCC on the indexed search log analysis

benchmarks, taking 0.90x and 0.93x less time for the single and multithreaded versions, respectively. Further, V8 has the same performance as GCC on the single-threaded regular expression based log analysis. In both cases, the good performance comes from the library implementation of pointer copying (in the case of Go on indexed log analysis) and the regular expression engine (in the case of V8 on regular expression based log search).

Surprisingly, whereas Go spends a total of 0.04 seconds in a critical section in indexed log search, GCC takes 2.49 seconds. In the log analysis benchmarks, each worker thread returns a list of matched log messages to the main thread by appending a thread local list of results to the main thread's global list, while holding a lock. Inside this critical section, for the append operation, Go copies more pointers per loop iteration than GCC.

In Go, appending to a list (referred to as a slice) is done efficiently because slices are an intrinsic type and appending is performed by a builtin function, which uses hand written assembly. Both GCC and Go use 128-bit wide XMM registers [9] to move two 64-bit pointers at a time with a single `mov` instruction. The assembly in Go unrolls the loop as much as possible, using all 16 XMM registers to move 32 64-bit pointers per iteration. Furthermore, rather than checking if a write barrier is required for each pointer, Go checks once if write barriers are required for all pointers, as they do not need to be performed when concurrent marking is not active⁸.

For the same operation in C++ with `std::vector`, GCC only moves 2 64-bit pointers in a single XMM register per iteration. Therefore, for every 2 pointers (or single XMM register move) GCC must also execute a compare and jump instruction to iterate the loop. On the other hand, Go will only execute these two loop iteration instructions every 32 pointers (16 XMM register moves). Furthermore, because we use `std::unique_ptr`, GCC must set the pointers in the thread local vector to `NULL`, as ownership has been transferred to the global vector. GCC stores `NULL` to 2 pointers each iteration of the loop using another XMM register.

⁸Write barriers are explained in more detail in Section 6.3.



Automatic Recovery of Fine-grained Compiler Artifacts at the Binary Level

Yufei Du¹, Ryan Court², Kevin Snow², and Fabian Monroe¹

¹University of North Carolina at Chapel Hill

²Zerpoint Dynamics

Abstract

Identifying a binary's compiler configuration enables developers and analysts to locate potential security issues caused by optimization side-effects, identify binary clones, and build compatible binary patches. Existing work focuses on identifying compiler family, version and optimization level of a binary using semantic features and deep learning techniques. Unfortunately, in practice, binaries are an amalgamation of objects and functions that can be compiled at different optimization levels with a variety of individual, fine-grained, optimizations that may be applied depending on the structure of the code. Hence, rather than recovering high-level artifacts, i.e., compiler family, version, and optimization level, we explore the recovery of individual, fine-grained, optimization passes for each function in a binary. To do so, we develop an approach using specially crafted features alongside intuitive and understandable machine learning models. Our evaluation on 15 popular open-source repositories shows that our approach compares favorably with the state-of-the-art deep learning approach in compiler family, compiler version and optimization level identification. For fine-grained optimization passes, our evaluation on 149,814 functions from 552 binaries in four popular open-source repositories shows that our approach achieves an average F-1 score of 92.1% for all optimization passes and an average F-1 score of 89.8% for optimization passes that could have negative impacts on security. Moreover, our approach includes experimental support for dynamic feature extraction via binary emulation, and our results show that such features offer promising potential in improving the accuracy of optimization pass identification.

1 Introduction

Modern compilers serve much more than simple translators that convert human-readable program code to machine instructions. They are also fully automated optimizers that improve the performance of code via numer-

ous translations, including the removal of unused code, re-ordering of instructions, replacing expensive computations with more efficient ones, and merging functions. Ideally, these optimizations do not change the behavior of the program in any way, other than making the resulting code faster and smaller.

However, while compiler optimizations are performed in ways that should not interfere with the normal execution of a program, some optimizations could have a negative impact on security. In fact, recent studies [3, 19, 21, 22, 24] have shown that certain optimizations could nullify protections and verification of secure functions as well as introduce timing side channels. For example, the dead code elimination optimization could remove instructions that erase sensitive data after using it, causing the sensitive data to be vulnerable to leakage if there is a memory error later on in the program. Similarly, the strength reduction optimization that replaces expensive operations with more efficient ones could open side channels. Developers are usually unaware of these optimizations because the compiler automatically chooses the set of optimizations to apply based on the optimization level, code structure, target architecture, and target processor family. Moreover, even subtle changes can create additional code reuse gadgets, causing the compiled program to be more vulnerable to attacks [1].

One way to avoid these pitfalls is by having developers manually tweak their compilation scripts to avoid applying potentially risky optimizations to secure functions that handle sensitive data. However, for large projects, this would be a daunting and tedious task. In order to maximize performance while avoiding risky optimizations, developers need to manually tweak optimization flags in addition to the optimization level. With hundreds of both architecture-independent and architecture-dependent flags, manually tweaking optimization flags is challenging and time-consuming [5]. Moreover, some compilers also include hidden optimizations that cannot be manually controlled, and the code base for mod-

ern compilers is too large for users to review and study for the logic behind optimizations [7]. In practice, developers for security-critical projects, such as OpenSSL and mbed TLS, take another approach by implementing workarounds in the source code of secure functions to “confuse” the compiler such that it does not apply optimizations to the secure code [19, 21, 24].

These workarounds, however, are not always stable. As compilers introduce more optimizations in each release, a workaround that successfully tricks one version may not be effective for a future version, and developers then need to implement a more complex workaround [19]. Therefore, a solution that helps identify optimizations applied to a binary at the function level could offer significant practical value: developers could use such an approach to, for example, verify that the program is compiled using optimizations that do not negatively impact security before releasing the binary.

In addition to safety verification, compiler optimization classification could be beneficial to binary code clone detection and binary patching. Compiler configurations can cause significant degradation in the performance of clone detection techniques [6, 8, 10, 12]. Similarly, in binary patching [4], locating the vulnerable function to be patched becomes more difficult in the presence of certain compiler optimizations.

Facing some of these challenges ourselves when trying to safely perform binary patching at a function level, we revisited the state of the art in compiler artifact recovery. We found that the most comprehensive of these techniques [2, 16, 17, 20] focus on identifying the compiler family, the major compiler version, and the optimization level for a binary, either for individual functions or for the entire binary — but, we need more detailed information (i.e., the passes that may have been applied). Unfortunately, this level of recovery has not been well explored, and even when it has been mentioned, the authors conclude that “[f]urthermore, some flags would be challenging, if not impossible to detect, the dead code elimination flag being one example” [15].

Additionally, we found that contemporary approaches rely on either semantic features (e.g., the control flow graph) or employ deep learning. As the interpretability of the outputs was a key motivating factor for us, we instead opted for the use of shallow learning with specially crafted features. To our surprise, our approach performed on par with or better than the state of the art [20] that uses highly-tuned neural networks for optimization level identification (e.g., `-O1` versus `-O3`). More importantly, we take a step further and show that contrary to recent statements by Pizzolotto and Inoue [15], one can detect the application of certain optimization passes with good accuracy. Our approach, coined *PassTell*, helps identify optimization passes that affect security for individual

functions, such as different forms of dead code elimination, code motion, and strength reduction passes.

Our specific contributions include:

1. We designed *PassTell*, a new approach in compiler configuration identification that recovers the optimization passes applied at the function level.
2. We explored the effects of using dynamic features extracted by force-executing each function. We show that the use of such features offers potential in improving accuracy, albeit in certain cases.
3. We evaluated our machine learning approach and compared the results with the state-of-the-art. We find that our approach performs on par with the state-of-the-art in identifying compiler family, compiler version, and coarse-grained optimization level.
4. We evaluated our approach using four variants of 138 programs from four open source repositories built with the latest development version of the Clang 14 compiler. Our approach is capable of identifying most optimization passes with high accuracy.

2 Background

2.1 Compiler Optimization

Modern compilers (e.g., GCC, LLVM, and ICC) offer complex optimizations that improve the performance and reduce the code size of the compiled program. In theory, these compilers provide different levels of optimizations, and programmers only need to specify an optimization level for the compiler to automatically apply the corresponding set of optimizations.

In practice, however, optimization is more complicated than simply applying a fixed set of optimizations passes for each optimization level. The LLVM compiler [11], for example, uses pass managers to control the passes to apply as well as the order of running the passes. The pass manager considers multiple factors when deciding the passes to run in addition to the optimization level specified by the user, including the target architecture, the target processor generation, and the source code structure. For example, for programs targeting outdated x86 processors, the compiler would avoid applying optimizations that use the AVX instructions that were recently introduced, and for functions without loops, the pass manager would avoid running optimizations that improve loop performance. Therefore, knowing the optimization level of a binary is *not* enough to determine the exact set of optimizations applied to the binary.

2.2 Security Implications

While compilers can take care to ensure that their optimizations do not change the behavior of normal program execution, fully automated reasoning about the purpose of deliberately ineffective or seemingly useless operations added by the developers is not (yet) possible. Hence, such instructions are targets for optimization, potentially undermining security assumptions. D’Silva et al. [3] called this problem the “correctness-security gap” and defined three types of security violations caused by compiler optimization: persistent state, side channel attacks, and undefined behavior.

A persistent state violation is when data persists outside of the scope it is designed to be available. D’Silva et al. [3] listed three optimizations that could cause this violation: dead code elimination, function inlining, and code motion. For example, in a password verification function where the password is temporarily stored in the memory during verification, the compiler may consider the operations that erase the local memory to be dead code and remove them, causing the password to exist in the memory after it is used, until it is eventually overwritten by a later function. Similarly, if a trusted security-sensitive function is inlined in an untrusted function, then the lifetime of the local variables of the trusted function would be extended to when the untrusted function returns. Finally, code motion may switch the order of instructions to avoid unnecessary computation or to improve locality. This optimization may cause the program to write sensitive values to memory before verifying that the operation is needed. Beside these three optimizations, Simon et al. [19] added that in situations where the entire stack frame needs to be erased, any optimization that changes the size or the layout of the stack frame such as tail-call optimizations may cause the erasure to be incomplete.

Compiler optimizations could also introduce side-channels that leak information about the program’s execution based on its timing or memory usage. To avoid side-channels, the developer may add unnecessary or inefficient operations to functions, but the optimizations may simplify or remove these operations, thereby reintroducing the side-channel. D’Silva et al. [3] listed three optimizations that could introduce side-channels: common subexpression elimination, which merges multiple instructions into one instruction to avoid duplicate computation; strength reduction, which replaces expensive instructions with more efficient ones, and peephole optimization, which inspects surrounding instructions to find opportunities to reorder or replace instructions for simpler computation or better locality.

Our work focuses on identifying optimizations that could cause persistent state violations or side-channels. Identifying undefined behavior (i.e., violations caused by

undefined behavior when developers use semantics that are undefined by the specification) is out of scope.

3 Related Work

Rosenblum et al. [18] presented seminal work in the area of compiler identification from binary files. Their approach focuses on identifying only the compiler family for code snippets in the IA-32 architecture using a probabilistic graphical model. Later on, Rosenblum et al. [17] extended that work and presented Origin, a tool that identifies compiler family, compiler version, and optimization level for each function in a binary. Origin uses a linear support vector machine model with features including idioms of instructions, sub-graphs of the control-flow graph, and high-level layout of functions such as the starting address. However, for optimization level, Origin could only perform coarse-grained identification with two options: “low” for -O0, -O1, and “high” for -O2, -O3.

A few years later, Rahimian et al. [16] presented a different approach (called BinComp) for compiler provenance identification. BinComp focuses on identifying the compiler family, compiler version, and optimization level for the entire binary. Different from Origin, BinComp heavily utilizes features extracted from utility functions added by the compiler to identify the compiler version and optimization level. These utility functions include program initialization, the startup code, and the termination code. While these functions could be highly indicative of an optimization level, it is impossible to perform identification for each function in a binary. Therefore, BinComp could only identify the compiler configuration used to compile the main routine of a program.

More recent work [2, 15, 20, 23] in compiler provenance identification started using neural networks for classification. Chen et al. [2] presented HIMALIA, a classifier using recurrent neural network to identify the optimization level for each function of a binary. HIMALIA uses vectors of disassembled instructions as features and uses two recurrent neural networks for classification. One network classifies the function into one of the four classes: -O0, -O1, -O2/O3, and -Os; the other network then differentiates -O2 and -O3. While the evaluation of HIMALIA includes binaries compiled with different versions of the LLVM Clang compiler, it only focuses on identifying the optimization level of each function, making no distinction of optimizations applied in different compiler versions. Yang et al. [23] presented BinEye, a classifier using convolutional neural network to identify optimization levels for each object in ARM binaries. Since each instruction in the ARM architecture is four bytes, BinEye uses the first 1024 instructions of each object as raw features and extracts word and position embeddings from them. Tian et al. [20] presented NeuralCI,

a classifier with either convolutional neural network or recurrent neural network to identify the compiler family, compiler version, and optimization level for each function in a binary. NeuralCI uses Word2Vec [14] embedding to allow instructions with variable size. Similar to BinEye, the evaluation of NeuralCI combined `-O2` and `-O3` into one coarse-grained optimization level of `OH`.

Most recently, Pizzolotto and Inoue [15] presented an approach that uses either a convolutional neural network or a long-short term memory network to identify the compiler family and optimization level for code snippets of 2KB in seven different architectures. This approach includes either the raw bytes or the opcodes as features but concluded that raw bytes lead to better results when large amounts of training data are available. Similar to HIMALIA, the evaluation of this approach includes five different optimization levels: `-O0`, `-O1`, `-O2`, `-O3`, and `-Os`. As NeuralCI performed the most in-depth and realistic evaluation, and it was shown to outperform the other approaches that classify at the function level, we select it for comparison later on in this paper.

4 Approach

We now present PassTell, an approach for identifying the set of optimization passes likely applied to each function in a binary file. Figure 1 shows the overall workflow.

4.1 Dataset Generation

Our dataset used to train the classifier includes functions compiled with different optimizations. Each function includes a set of optimization passes that were applied during compilation and the instructions in the function. The overall workflow of dataset generation is as follows: first, we compile programs with different optimization levels and record the list of optimizations applied to each function; then, we disassemble the binary to retrieve the instructions of each function; finally, we sanitize the instructions by removing detailed memory addresses, call targets, and immediate values.

To train and evaluate our classifier, we first need to gather the optimization passes that modify a function during compilation. While the Clang frontend of the LLVM compiler [11] has an option to list the optimizations applied to each function during compilation, this option outputs all passes that run, even the ones that make no modification. Therefore, we made modifications to extract the optimization passes that modify a function during compilation. Section 5 discusses the modifications we made to the LLVM compiler.

After disassembling the binary file, we sanitize the instructions before feature extraction. Specifically, we replace all memory addresses with the `#MEM#` label, all

call targets with `#TARGET#`, and all immediate values with `#IMM#`. We make this adjustment because the detailed memory addresses, immediate values, and call targets are highly variable across different programs and are not useful in optimization classification. Tian et al. [20] applied similar rules to the instructions.

4.2 Dynamic Feature Generation

As an extension, we also present a method to extract changes of register values (recovered via emulation) and use these dynamic features in our classifier. At present, the dynamic features only include register deltas. We use a binary emulation library to attempt to force-execute each function in the dataset. After the execution of each instruction, we record the address of the instruction, the registers changed, and the deltas of their values. Within these three types of data, the instruction address is only used to compute the coverage of the force-execution and to avoid endless loops.

While it may seem unnecessary to include dynamic features as the dataset already includes the entire disassembly of the function, we posit that dynamic features could still contribute to the classification because some changes in the registers are implicit and not shown in the disassembly. For example, the `FPSW` register includes flags, the stack address, and the current code for floating point operations. Additionally, floating point operations may cause this register to change implicitly. Our main goal of including dynamic features is to explore their potential to improve classification accuracy. We expect progress can still be made in future work.

4.3 Feature Extraction

Category	Feature Type	Example
Static	Opcode	<code>call</code>
	Instruction	<code>mov esi ecx</code>
	Register	<code>rsi</code>
	2-gram of opcodes	<code>pop ret</code>
	2-gram of instructions	<code>pop rbp ret</code>
	First instruction	<code>push r15</code>
Dynamic	Last instruction	<code>xchg ax ax</code>
	Register value delta	<code>rbp=-248</code>

Table 1: Feature types and examples for each type of feature used in our approach

By default, we use seven types of static features: opcode, instruction, register, two-gram of opcodes, two-gram of instructions, and the first and last instruction of a function. The register value delta is an optional feature. Table 1 lists an example of each feature type.

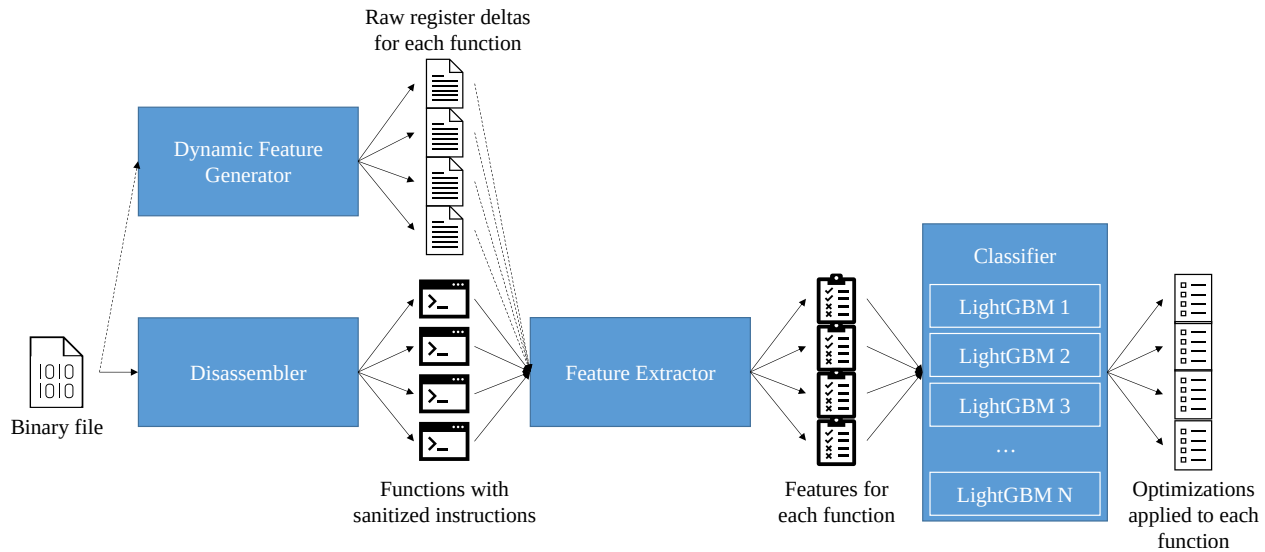


Figure 1: Workflow of PassTell for optimization pass classification

We apply feature selection before extracting features. There are overwhelming amount of different instructions, two-grams of opcodes, two-grams of instructions, and changes of register values, so we select 1,000 features from each of these four feature types, for each optimization pass. This means that we use different features for the classification of each optimization pass. Our feature selection strategy for each optimization pass is as follows: first, we filter the dataset such that it is balanced for the optimization pass (i.e., the number of functions with the optimization and the number of functions without the optimization are the same); second, we sort the features by the number of functions in the balanced dataset that have at least one occurrence of the feature; finally, we select the 1,000 most frequent features of each type. Table 2 shows an example of the selected features for the `Early CSE` pass. For the other feature types, including opcode, register, the first instruction, and the last instruction, we use all features in these types without applying feature selection because there are less features in these types.

All features hold binary values. That is, we check whether the function has this feature or not. For example, a leaf function (i.e., a function that does not call any other functions) that performs arithmetic operations in a loop should have value 0 for the opcode feature `call` because it does not include any call, and it may have value 1 for the opcode feature `test` because it may use the `test` instruction to determine the end condition of the loop.

We decide to use this simple set of binary features as a result of interpretable feature engineering. Our approach with interpretable feature importance allows us to compare and select the most efficient and effective feature types. We also tested using frequencies as features instead of binaries but found minimal improvements.

4.4 Classification

After generating the feature set, we then train our classifier for optimization classification. For each optimization, we train a binary LightGBM classifier [9] that decides if a function is modified by this single optimization.

LightGBM is an implementation of gradient boosting decision tree. We select this classifier instead of deep learning techniques for two reasons. First, modern compilers include a large amount of passes. In our dataset, for example, we observed a total of 83 unique compiler passes applied to functions or components inside a function (e.g., a loop). Each pass requires a separate binary classifier. Therefore, in order to train the dataset with a reasonably large dataset, our choice of classifier should scale well in both training time and memory consumption. Second, classifiers such as support vector machines and neural networks cannot easily demonstrate the reasoning or the importance of features. Decision tree-based classifiers, on the other hand, can more readily show the reasoning and the importance of features.

In the training phase, a list of all optimization passes is generated and a model for each is created. That is, for each function, we create a list of binary labels for each pass to indicate if the function was modified by each of these corresponding passes, then a LightGBM-based classifier is trained for each of those binary labels (i.e., 83 classifiers in total) using the extracted features. For classification, the feature extractor sends each function's features to each of the 83 trained models. The trained model for each optimization pass then decides if that optimization was applied to (and modified) each function, and finally the list of all applied optimization passes is returned for each individual function in the binary.

Static				Dynamic			
Instruction		2-gram of Opcodes		2-gram of Instructions		Register Value Delta	
Feature	Count	Feature	Count	Feature	Count	Feature	Count
<code>ret</code>	2,626	<code>mov; mov</code>	2,577	<code>pop rbp; ret</code>	1,517	<code>ip=3</code>	2,710
<code>call #T#</code>	2,300	<code>pop; ret</code>	2,413	<code>push rbp; mov rsp rbp</code>	1,059	<code>rip=3</code>	2,710
<code>add #I# rsp</code>	1,688	<code>push; mov</code>	2,069	<code>add #I# rsp; pop rbx</code>	841	<code>eip=3</code>	2,710
<code>push rbp</code>	1,572	<code>mov; call</code>	2,059	<code>mov rsp rbp; sub #I# rsp</code>	814	<code>rsp=-8</code>	2,582
<code>pop rbp</code>	1,527	<code>call; mov</code>	1,680	<code>add #I# rsp; pop rbp</code>	788	<code>spl=-8</code>	2,582

Table 2: Top five selected features for feature types with feature selection for optimization pass `Early CSE` with a sample set of 2,780 functions, including 1,390 positive samples and 1,390 negative samples. (Sanitized keywords such as `#TARGET#` are abbreviated to only the first letter.)

5 Implementation

We now discuss the implementation details of the various components shown in Figure 1.

Dataset Generation We implemented the dataset generation component in Python. The technique takes a source code repository and compiles it with four levels of optimizations (`-O0`, `-O1`, `-O2`, and `-O3`). During compilation, it extracts the ground truth of optimization passes that modified each function from the compiler log. After compilation, `objdump` is used to disassemble the binary in order to retrieve the instructions of each function. Instructions are sanitized by removing detailed memory addresses, call targets, and immediate values (as discussed in Section 4.1).

We modified the legacy pass manager of the LLVM compiler [11] in order to retrieve the list of optimization passes that modifies a function. We made modifications to the latest development version of LLVM 14 at the time of this writing (commit `#c59ebe4`). We added an option to the existing output flag of LLVM’s legacy pass manager¹ to list the optimization passes that modify a function or components inside a function (e.g., a loop).

Finally, since we use `objdump` to disassemble the compiled binary files into functions, our tool enables debugging symbols during compilation. However, this is not a hard requirement for classification as one could use external tools such as IDA Pro to determine function boundaries of stripped binaries, but prior efforts found little difference in outcome between the two tactics [20].

Dynamic Feature Generation As mentioned in Section 4.2, `PassTell` also supports register values extracted

¹Recent versions of the LLVM compiler contains two pass managers, and by default, the new pass manager is responsible for all optimization passes before code generation. However, the new pass manager does not contain any utility functions to extract the pass names. Therefore, we use the flag `--flegacy-pass-manager` to use the legacy pass manager for all passes, including the ones before code generation.

during execution to complement static features. To do this, we make use of Zelos [25], a python-based binary emulator platform that supports x86 (both 32 and 64-bit), ARM and MIPS architecture emulation. Under the hood, Zelos makes use of QEMU CPU emulation and implements system call emulation similar to QEMU usermode, but CPU and syscall-level hooks make comprehensive binary instrumentation readily available. Using this tool, we instrument forced emulation of each function in a binary and record the changes to register values after each instruction within function boundaries. To do so, we extended the emulator to execute a binary, then wait until it has mapped itself in memory and pause execution. Then, for each function, the instruction pointer is adjusted to the function start address before execution resumes. While executing the function, `call` instructions are skipped to ensure that recorded register changes reflect only the target function, while also avoiding recursion and potentially long call chains. Before emulating each function, we map a page of memory and fill it with the start address of the region. The address of this region is used as the return address for the target function as well as for all existing register values that appear to be pointers to memory, to avoid errors related to reading or writing unmapped memory.

Feature Extraction and Classification As discussed in Section 4.3, our approach performs feature selection for each optimization pass. As such, having a dedicated feature extraction component that saves all features to files is highly inefficient. Therefore, we combined the feature extraction phase for both static and dynamic features and the classifier into one classifier component.

We implemented our classifier using Python and the LightGBM library [13]. The classifier iterates through each optimization pass, selecting and extracting features and then creating a `LGBMClassifier`. When training, the classifier first filters the dataset such that the dataset is balanced, with the same amount of positive and negative data. Then, the classifier selects the most popular

features as described in Section 4.3, extracts both static and dynamic features, and trains the `LGBMClassifier`. When classifying, the classifier extracts the static and dynamic features and uses the `LGBMClassifier` to predict whether the function is modified by this optimization pass. After the classifier extracts features and performs classification for all optimization passes, it then merges the results together to generate the final result, the list of optimizations applied to the function.

6 Evaluation

The evaluation includes experiments in two directions. First, we evaluate the effectiveness of our features and our classifier. In this experiment, we compare PassTell with NeuralCI [20], a state-of-the-art approach to compiler configuration identification. To ensure a fair comparison, we first modified our classifier into a multi-class classifier to identify the same compiler configuration as NeuralCI, including the compiler family, the major compiler version, and the optimization level, where each combination is a class. Later, we evaluate our approach in identifying the individual optimization passes. Overall, our experiments seek to answer the following questions:

- RQ1** *How does our approach compare to the state-of-the-art in compiler configuration recovery?*
- RQ2** *Can our approach provide meaningful information regarding feature significance?*
- RQ3** *How well can we infer individual optimization passes?*
- RQ4** *Does the inclusion of dynamic features help with classifying individual passes?*

6.1 Compiler Configuration Identification

We use the same benchmark programs as in NeuralCI [20]. To replicate NeuralCI’s experimental setup, we combine binaries compiled with `-O2` and `-O3` optimization levels into one class, `-OH`. As our prototype utilizes `objdump` to generate the disassembly for each function, we only use the unstripped binaries. Tian et al. [20] observed no difference in classification performance between the stripped and unstripped binaries. Our dataset thus consist of all dynamically linked unstripped executables from the dataset, including `binutils`, `busybox`, `coreutils`, `curl`, `ffmpeg`, `git`, `gsl`, `libpng`, `openssl`, `postgresql`, `sqlite`, `valgrind`, `vim`, `zlib`, and `gdb`.

While inspecting the dataset, we discovered that the dataset is highly unbalanced, with some configurations having significantly less amount of samples than others. We note that this issue is not limited to NeuralCI, as other

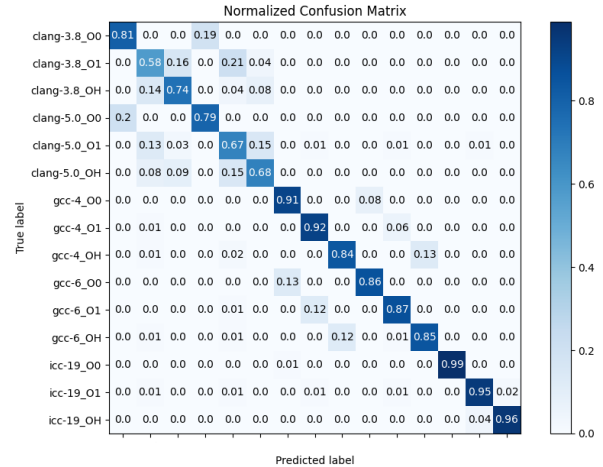


Figure 2: Confusion matrix of our LightGBM model applied to the NeuralCI formulation of the compiler classification problem

approaches [2, 15] also do not balance their datasets. To circumvent this issue, we randomly dropped functions from certain configurations to ensure that all configurations have the same number of functions. In the end, our dataset consists of 4,400 functions for each configuration. We split the dataset into an 80% training set and a 20% testing set. Appendix A describes the issue in detail and includes a comparison of the results of NeuralCI before and after balancing the dataset. Finally, NeuralCI includes only unique functions in its dataset, so functions that are identical across configurations are removed. We apply the same procedure.

Since Tian et al. [20] do not include all the code used to construct features from the dataset, we re-implemented the extraction and abstraction of functions. For extraction, we use `objdump` instead of IDA Pro to parse the body of each function. The abstraction for each function is the same as done by Tian et al. [20], namely, mnemonics and register operands are unchanged, base memory addresses in operands are replaced with the symbol `#MEM#`, and immediate values are replaced with the symbol `#IMM#`.

Experiment Results

For the identification of compiler family, compiler version, and optimization level, NeuralCI achieves an average F-1 score of 76.6%, and our approach achieves an average F-1 score of 83.2%. Our re-implementation of NeuralCI produces lower results as reported in their paper [20]. We attribute the variation to be due to the correctly balanced dataset. Overall, the results show that our approach performs better than NeuralCI in identifying the compiler family, the major compiler version, and optimization level.

Interestingly, both the confusion matrix of our ap-

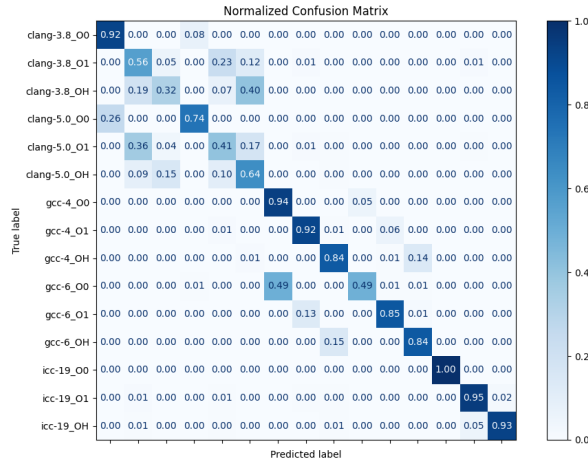


Figure 3: Confusion matrix of NeuralCI in classifying the compiler family, compiler version, and optimization level

proach (Figure 2) and the confusion matrix of NeuralCI (Figure 3) show that identifying the optimization level and the compiler version of binaries compiled with Clang is more challenging. NeuralCI reports similar findings in its evaluation [20]. For functions built by GCC and ICC, both approaches achieve high accuracy in identifying the compiler family, compiler version, and optimization level. For this reason and the fact that we modified the LLVM pass manager to extract pass information, we focus on Clang² for the remaining experiments.

6.2 Optimization Pass Identification

Satisfied with the performance and simplicity offered by PassTell, we focused on tackling the more difficult cases with Clang. Specifically, the compiler pass dataset consists of functions from `binutils` (2.37), `coreutils` (9.0), `httpd` (2.4.51), and `sqlite` (3.36.0) programs compiled with Clang 14, using each of `-O0`, `-O1`, `-O2`, and `-O3` optimization levels, generating a total of 149,814 functions in 552 binaries. Then, we balance the dataset for each pass: for each pass, we randomly select an equal amount of functions with the pass applied (i.e., positive samples) and functions without the pass applied (i.e., negative samples). We also limit the maximum number of samples for each pass to 5,000 positive samples and 5,000 negative samples.

Experiment Results

Overall, our approach achieves an average F-1 score of 92.1%. All but three of the 83 passes have an F-1 score higher than 80%, and the three exceptions all have insufficient amount of samples (<150 functions). If we only con-

²Clang is a front-end of LLVM and is part of the LLVM infrastructure.

sider the 73 passes that contain more than 500 samples, the average F-1 score improves to 93.7%. Table 3 depicts the results of our approach using only static features. For brevity, we only list the detailed results of 13 of the 83 passes in total. We pick these 13 passes because they are optimizations that could affect security [3, 19, 21, 24]: dead store elimination, dead code elimination, code motion, tail call optimization, common subexpression elimination, strength reduction, and peephole optimizations. For these passes, our approach achieves an average F-1 score of 89.8%. The findings further show that, contrary to Pizzolotto and Inoue [15]’s statement, even passes that seem unlikely to be detected, such as dead code elimination, can be identified with high accuracy.

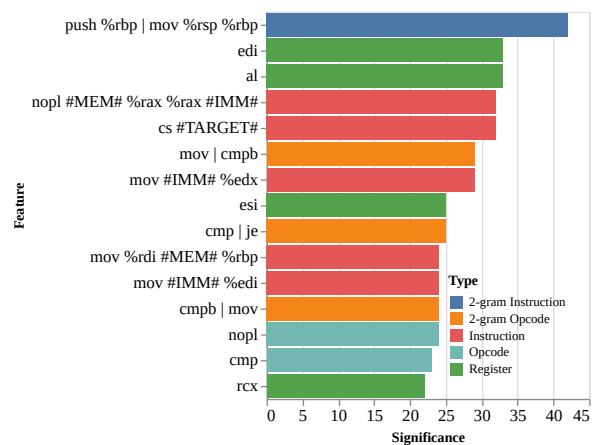


Figure 4: Top 15 Features for Aggressive Dead Code Elimination

To understand why our approach works as well as it does, we inspected the feature significance of some of the security-affecting passes. We choose to inspect the feature significance for Aggressive Dead Code Elimination and Peephole Optimizations because these two passes offer sufficient descriptions about their purposes in the code comment of the LLVM compiler’s source code. Figure 4 shows the top 15 features for the Aggressive Dead Code Elimination pass and the feature type of each feature. The description of the optimization pass in the LLVM compiler’s source code indicates that this pass considers all code to be dead unless proven otherwise and removes all the dead instructions, especially dead code involving loops. The top feature in this case checks whether the function updates the function pointer. Clang omits updating the frame pointer on optimization levels above `-O1`. Similarly, the instruction features `nopl #MEM# %rax $rax #IMM#` and `cs #TARGET#`³ are instructions padding instructions without any semantic meaning whose purpose

³This feature is actually a `nopw` NOP instruction. Due to the different format `objdump` uses for instructions involving the `cs` segment register, this instruction is not parsed correctly. Since segment registers

Pass	Training Samples	Testing Samples	Precision (%)	Recall (%)	F-1 (%)
Dead Store Elimination	1332	444	86.3	85.8	85.7
Aggressive Dead Code Elimination	1092	364	83.9	83.5	83.4
Bit-Tracking Dead Code Elimination	2512	838	87.6	87.5	87.5
Remove dead machine instructions	7500	2500	88.7	88.6	88.6
Early Machine Loop Invariant Code Motion	7500	2500	93.4	93.3	93.3
Machine Loop Invariant Code Motion	739	247	89.4	89.0	89.0
Loop Invariant Code Motion	7500	2500	90.8	90.6	90.6
Tail Call Elimination	88	30	86.6	86.6	86.6
Machine Common Subexpression Elimination	7500	2500	88.5	88.1	88.1
Early CSE	7500	2500	92.5	92.2	92.2
Early CSE w/ MemorySSA	7500	2500	88.9	88.6	88.6
Loop Strength Reduction	7500	2500	95.4	95.4	95.4
Peephole Optimizations	7500	2500	98.0	98.0	98.0
Average			90.0	89.8	89.8

Table 3: Precision, recall, and F-1 results on security-related passes when using static features.

is to enforce a 16-byte alignment between functions, and Clang only adds these instructions at `-O1` or above. Since Aggressive Dead Code Elimination is never applied at `-O0`, these three features effectively remove functions with `-O0` optimization level.

Four other features among the top 15 features include various forms of the `cmp` compare instruction, which commonly appears in loops. This finding matches the description of the pass. Finally, some of the top features show significance in the usage of certain registers. Since this pass is applied before register allocation in the compiler pipeline, we speculate that the removal of dead instructions reduces the amount of required registers, causing the register allocator to not use certain registers.

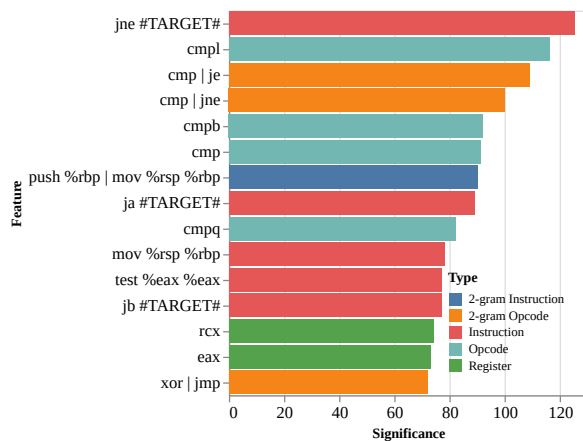


Figure 5: Top 15 Features for Peephole Optimizations

The top features for Peephole Optimizations show similar patterns. Figure 5 shows the top 15 features. The description in the code comments suggests that

are rarely used in 64-bit x86 programs, and we did not find any instructions utilizing that register other than the padding NOP, we conclude that this small implementation quirk does not impact our results.

this pass performs four types of optimizations: optimization of sign/zero extension instructions, optimization of comparison instructions, optimization of loads, and optimization of copies and bitcasts. The top features show that our classifier mainly capture the second type of optimizations that optimizes comparison instructions. Six of the 15 features include a variant of the `cmp` compare instruction; four other features include a jump instruction that usually follows a compare instruction, and one feature includes a `test` instruction, which is functionally similar to a compare instruction. Similar to Aggressive Dead Code Elimination, the top features for Peephole Optimizations also include the feature that saves the frame pointer in order to detect `-O0` functions.

6.3 Case Study on Optimization-induced Vulnerabilities

Although prior studies [3, 19, 21, 22, 24] have shown that compiler optimizations can weaken protections put in place by safe coding practices, it remained unclear whether the nullification of protections could in fact lead to information leakage or other attacks for real-world programs. Thus, we studied three real-world programs and examined how protections introduced by programmers are affected by the dead store elimination optimization: BusyBox (1.35.0), `httpd` (2.4.52), and `crypto++` (5.6.4).

BusyBox BusyBox is a popular embedded program that combines many UNIX utilities into a single binary. Some of the utilities included in BusyBox require password encryption or authentication, such as the `passwd` utility, the `cryptpw` utility, the built-in HTTP server, and the built-in FTP server. Therefore, BusyBox’s codebase includes hashing functions such as MD5 and SHA. When a utility needs to encrypt a password, the utility would call

`pw_encrypt`, which then calls the corresponding MD5, SHA, or DES encryption functions. The SHA encryption function, `sha_crypt`, uses both a local stack object, `L`, and heap objects, `key_data` and `salt_data`, to store intermediate values during the encryption. Listing 1 shows a snippet of this function. Before the function returns, it attempts to erase the three objects using `memset`. However, at `-O3` optimization level, the dead store elimination optimization removes all three calls to `memset`. Thus, the intermediate values remain in the memory after the function returns even though developers took the necessary precautions to erase the data.

```

1 static char * NOINLINE sha_crypt(/*const*/
   char *key_data, /*const*/ char *salt_data)
2 {
3     ...
4     struct {
5         ...
6     } L __attribute__((__aligned__((__alignof__(
       uint64_t)))));
7     ...
8     salt_data = xstrndup(salt_data, salt_len);
9     ...
10    key_data = xstrdup(key_data);
11    ...
12    /* Clear the buffer for the intermediate
       result so that people
13     attaching to processes or reading core
       dumps cannot get any
14     information. */
15    memset(&L, 0, sizeof(L));
16    memset(key_data, 0, key_len);
17    memset(salt_data, 0, salt_len);
18    free(key_data);
19    free(salt_data);
20    ...
21    return result;
22 }

```

Listing 1: Code snippet from the `sha_crypt` function

We tested this program using its `cryptpw` utility that prints the hashed password in the format of Linux's `passwd` format from a given password in plain text. The `cryptpw` utility calls `pw_encrypt` to hash the password text, which calls `sha_crypt` if SHA mode is selected. At `-O0` optimization level, all three objects that contains intermediate values in `sha_crypt` (`L`, `key_data` and `salt_data`) are overwritten to 0 properly at the end of the function. At `-O3` optimization level, however, the entire value of `L` is left on the stack at the end of the function. For the heap objects, the calls to `free` overwrite the first 16 bytes of both `key_data` and `salt_data`. Since `salt_data` is less than 16 bytes, its value cannot be recovered. However, in situations where the input plain text password is longer than 16 characters, the size of `key_data` would be larger than 16 bytes. In this case, part of the intermediate value stored in `key_data` would persist in the heap memory after its scope. After `sha_crypt` returns, `pw_encrypt` then calls a simple clean up func-

tion and returns to the caller utility. At this point, both the stack object `L` and the partial leftover from the heap object `key_data` still exist in the memory without being overwritten. This means that if an attacker finds a memory disclosure vulnerability in the caller utility of `BusyBox` before the intermediate values are eventually overwritten by subsequent functions, then the attacker can recover the entire value of `L` and/or the value of `key_data` after the first 16 bytes. Since `BusyBox` includes an HTTP server and an FTP server that both use `pw_encrypt`, it would be possible for an attacker to launch an attack remotely.

We have submitted a bug report for this issue ⁴.

```

1 int get_password(struct passwd_ctx *ctx)
2 {
3     char buf[MAX_STRING_LEN + 1];
4     ...
5     else {
6         ...
7         apr_password_get("Re-type new password
           : ", buf, &bufsize);
8         if (strcmp(ctx->passwd, buf) != 0) {
9             ctx->errstr = "password
               verification error";
10            memset(ctx->passwd, '\0', strlen(
                ctx->passwd));
11            memset(buf, '\0', sizeof(buf));
12            return ERR_PWMMISMATCH;
13        }
14    }
15    memset(buf, '\0', sizeof(buf));
16    return 0;
17    ...
18 }

```

Listing 2: Code snippet from the `get_password` function

Httpd and Crypto++ The HTTP server `httpd` contains a support utility program, `htpasswd`, that manages the files that store usernames and passwords. This program includes a function `get_password()` that reads the password entered by the user to a local buffer, and stores the buffer to a `passwd_ctx` struct. Before it returns, the function calls `memset` to erase the buffer such that the password entered by the user would not stay in the memory. Listing 2 shows a snippet of this function. Before any return statement, the function erases the memory of the buffer, `buf`, that contains the user-entered password at line 11 and line 15 in Listing 2. However, at `-O3` optimization level, the dead store elimination optimization removes the function call to `memset` that erases `buf`, causing the password in plain text to persist in the stack memory after `get_password()` returns, contrary to the developers' intention.

We tested this program and discovered that immediately after the function returns, we could recover the exact password in plain text in the stack memory. It is

⁴https://bugs.busybox.net/show_bug.cgi?id=14806

worth noting that after `get_password` returns to its caller, `mkehash`, the caller later calls other functions, overwriting the stack memory that contains the password as the stack frame of `get_password` is smaller than the stack frame of subsequent functions. Therefore, if an attacker finds a memory disclosure vulnerability in `mkehash` between the call to `get_password` and the subsequent function call, then they can retrieve the password in plain text.

`Crypto++`, a cryptography library written in C++, contains a similar issue as `htpasswd`. The function `CAST256::Base::UncheckedSetKey` uses a call to `memset` to erase its local variable, `kappa`, which holds the hashed key. At `-O3` optimization, the dead store elimination optimization removes this call, causing the secret key to remain in stack memory after the function returns.

For validation purposes, we use the entire compiler pass dataset (discussed in Section 6.2) as our training set for `BusyBox` and `crypto++`. For `httpd`, our compiler pass dataset also contains `httpd`, albeit a different minor version, so we exclude all functions in `httpd` from our training set. Our classifier correctly identifies the `Dead Store Elimination` pass for all three functions at `-O3` optimization level. At `-O0`, the classifier mis-identifies `sha_crypt` for `BusyBox` but identifies the missing of the pass correctly for `httpd` and `crypto++`.

6.4 The Effects of the Dynamic Features

Finally, we evaluate the value of including dynamic features. For that, we use the same dataset as in Section 6.2, but apply additional filtering. Specifically, because our current implementation of our dynamic feature generator is unable to generate register features for all functions in the dataset, we filter the dataset to only include functions that our generator achieved a minimum coverage threshold. Additionally, we only include passes that were applied to more than 500 functions. Since our goal is to evaluate the effects of the dynamic features and not the coverage of the feature generator itself, we view this as a reasonable way to understand what impact dynamic features could have on classification performance.

To compare to the baseline, we ran our approach using both static features and dynamic features. For the latter, we experimented with coverage thresholds ranging from 30% to 60%. For these configurations, we used the same filtered dataset and artificially removed dynamic features to reach the target coverage.

Experiment Results

Table 4 depicts the results. We only show cases where the difference in F-1 score is greater than 1%. While the average F-1 score appears similar regardless of the inclusion of dynamic features, the detailed results tell a more

compelling story. Some passes, such as `Remove dead machine instructions`, show a notable decrease in F-1 score, while others (e.g., `Early CSE w/ MemorySSA`) improve by almost 4.0%.

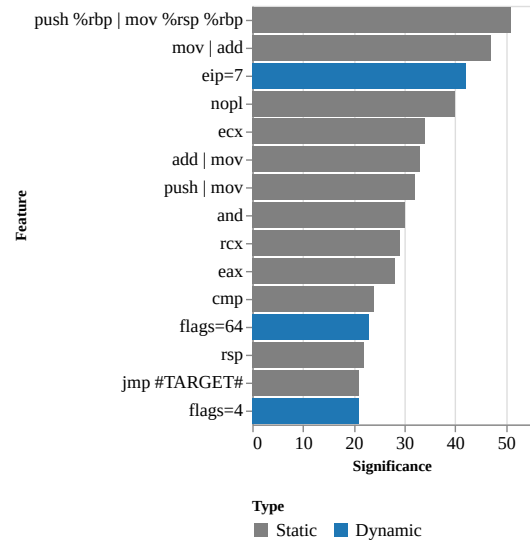


Figure 6: Top 15 Features for Early CSE w/ MemorySSA with both static and dynamic features

To better understand the reasons, we examined the most important features as deemed by the `LGBMClassifier` classifier. Figure 6 shows the top 15 features and the significance of each feature for the `Early CSE w/ MemorySSA` pass. The description of this pass indicates that this pass removes “trivially redundant instructions”. The top 15 features include three dynamic features: `eip=7`, `flags=64`, and `flags=4`. The EIP register is the instruction pointer, and the delta of this register shows information of instruction size, which is not available in the static features. Similarly, the `FLAGS` register, as partial alias of the `EFLAGS` register, contains various processor flags that are implicitly set during arithmetic operations and interrupts. Since this register is only set implicitly as a side effect of instructions, static approaches cannot extract this information by analyzing the disassembly or the binary code.

An examination of the `Rotate Loops` pass yields similar insights. Here, the top 15 features (Figure 7) include two dynamic features: `flags=-68` and `eip=10`. These results indicate that dynamic features, instruction pointer and processor flags in particular, can provide useful information in the detection of some optimization passes.

Lastly, we noticed that as we varied the coverage threshold, a few passes showed a noticeable improvement in F-1 score at higher coverage levels. For these passes, the result appears to be related to the number of dynamic features and their significance. For example, for `Early CSE w/ MemorySSA`, at 30% coverage, the top 25

Pass	Training Samples	Testing Samples	Static Only (%)	Static & Dynamic (%)
Early CSE w/ MemorySSA	607	203	82.2	86.1
Rotate Loops	400	134	84.2	86.5
Merge disjoint stack slots	598	200	94.4	96.5
Control Flow Optimizer	2,200	734	95.6	97.1
Canonicalize natural loops	396	132	87.9	89.4
Peephole Optimizations	1,297	433	94.9	96.3
Two-Address instruction pass	3,454	1,152	94.9	96.0
Remove Redundant DEBUG_VALUE analysis	1,515	505	88.1	87.1
PostRA Machine Sink	634	212	90.1	89.1
Live DEBUG_VALUE analysis	3,030	1,010	97.4	95.9
Machine Instruction Scheduler	1,303	435	93.3	91.2
Simplify the CFG	1,912	638	91.2	89.0
Remove dead machine instructions	903	301	91.0	86.7
Average for All Passes with at Least 500 Samples			92.9	93.0

Table 4: F-1 scores using only static versus static and dynamic features. The table list only results where the difference of F-1 score is $\geq 1\%$. The coverage threshold is set to $\geq 70\%$. Security-related passes are highlighted in dark grey.

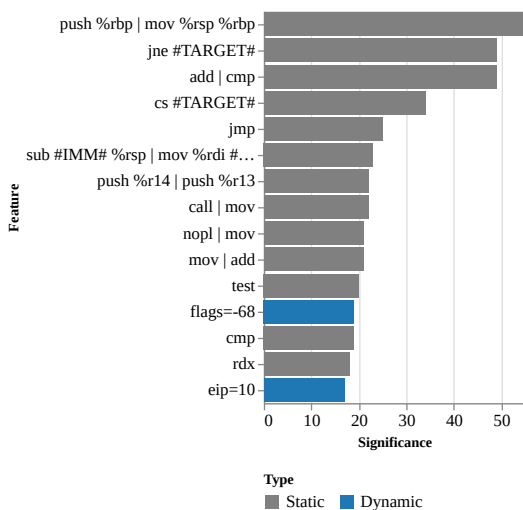


Figure 7: Top 15 Features for Rotate Loops with both static and dynamic features

features include four dynamic features, and at 70% or higher, the top 25 features include eight dynamic features. We posit that future improvements in the way dynamic features are collected could boost the classification for more passes, without negatively impacting others.

7 Limitations

Our implementation of the data collection component only records optimization passes applied to functions or components within a function (e.g., loops and basic blocks). Optimization passes applied to larger units such as modules and call graphs are not studied. Therefore, our analysis does not cover cross-function optimizations such as the function inlining. This limitation stems from the fact that the LLVM compiler does not report the specific

functions modified by a module pass or call graph pass. Similarly, our data collection component does not support extracting whole-program optimizations applied at link time, which could also negatively impact security [21].

In addition, our approach targets only binary files directly generated by the compiler. Thus, it can not be used in situations where the binaries are modified after compilation, such as obfuscated binaries or binaries with binary patches applied. This limitation is not unique to us.

8 Conclusion

We presented a light-weighted approach to the problem of compiler configuration identification. Our approach combines a novel technique for feature extraction and a scalable classifier for performing compiler provenance recovery. To further improve the accuracy of our classifier, we explored the use of dynamic features extracted by force-executing functions using a binary emulator. Overall, our approach shows comparable results as the state-of-the-art in the original problem of identifying the compiler family, the compiler version, and the optimization level, and pushes the field forward by showing that one can even identify individual optimization passes.

9 Availability

Our coarse-grained compiler configuration classifier and our fine-grained compiler pass classifier are available at <https://github.com/zeropointdynamics/passtell>. The balanced compiler configuration dataset (§6.1), the compiler pass dataset (§6.2), and the compiler pass dataset with high dynamic feature coverage (§6.4) are also available. Furthermore, Zelos, our binary emulator we use to generate dynamic features, is open sourced [25].

References

- [1] M. D. Brown, M. Pruet, R. Bigelow, G. Mururu, and S. Pande. Not so fast: understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets. *ACM Conference on Programming Languages*, 5:1–30, 2021.
- [2] Y. Chen, Z. Shi, H. Li, W. Zhao, Y. Liu, and Y. Qiao. Himalia: Recovering compiler optimization levels from binaries by deep learning. In *IntelliSys*, 2018.
- [3] V. D’Silva, M. Payer, and D. Song. The correctness-security gap in compiler optimization. In *IEEE Security and Privacy Workshops*, pages 73–87, 2015.
- [4] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee. Automating patching of vulnerable open-source software versions in application binaries. In *NDSS*, 2019.
- [5] K. Georgiou, Z. Chamski, A. A. García, D. May, and K. I. Eder. Lost in translation: Exposing hidden compiler optimization opportunities. *ArXiv*, abs/1903.11397, 2019.
- [6] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra. Finding software license violations through binary code clone detection. In *Conference on Mining Software Repositories*, pages 63–72, 2011.
- [7] M. J. Hohnka, J. A. Miller, K. M. Dacumos, T. J. Fritton, J. D. Erdley, and L. N. Long. Evaluation of compiler-induced vulnerabilities. *Journal of Aerospace Information Systems*, 16(10):409–426, 2019.
- [8] Y. Hu, Y. Zhang, J. Li, and D. Gu. Binary code clone detection across architectures and compiling configurations. In *International Conference on Program Comprehension*, pages 88–98, 2017.
- [9] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, 2017.
- [10] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned, 2021.
- [11] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, page 75, USA, 2004.
- [12] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 389–400, 2014.
- [13] Microsoft Corporation. Light gradient boosting machine. URL <https://github.com/microsoft/LightGBM>.
- [14] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
- [15] D. Pizzolotto and K. Inoue. Identifying compiler and optimization level in binary code from multiple architectures. *IEEE Access*, 2021.
- [16] A. Rahimian, P. Shirani, S. Alrbaee, L. Wang, and M. Debbabi. Bincomp: A stratified approach to compiler provenance attribution. *Digital Investigation*, 14:S146–S155, 2015.
- [17] N. Rosenblum, B. P. Miller, and X. Zhu. Recovering the toolchain provenance of binary code. In *International Symposium on Software Testing and Analysis*, pages 100–110, 2011.
- [18] N. E. Rosenblum, B. P. Miller, and X. Zhu. Extracting compiler provenance from program binaries. In *ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 21–28, 2010.
- [19] L. Simon, D. Chisnall, and R. Anderson. What you get is what you c: Controlling side effects in mainstream c compilers. In *IEEE European Symposium on Security and Privacy*, pages 1–15, 2018.
- [20] Z. Tian, Y. Huang, B. Xie, Y. Chen, L. Chen, and D. Wu. Fine-grained compiler identification with sequence-oriented neural modeling. *IEEE Access*, 9:49160–49175, 2021.
- [21] A. Venkatesh, A. B. Handadi, and M. Mory. Security implications of compiler optimizations on cryptography—a review. *arXiv preprint arXiv:1907.02530*, 2019.
- [22] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *ACM Symposium on Operating Systems Principles*, pages 260–275, 2013.

- [23] S. Yang, Z. Shi, G. Zhang, M. Li, Y. Ma, and L. Sun. Understand code style: Efficient CNN-based compiler optimization recognition system. In *IEEE Conference on Communications*, pages 1–6, 2019.
- [24] Z. Yang, B. Johannesmeyer, A. T. Olesen, S. Lerner, and K. Levchenko. Dead store elimination (still) considered harmful. In *USENIX Security Symposium*, pages 1025–1040, 2017.
- [25] Zeropoint Dynamics. Zelos, 2020. URL <https://github.com/zeropointdynamics/zelos>.

A The Unbalanced Dataset

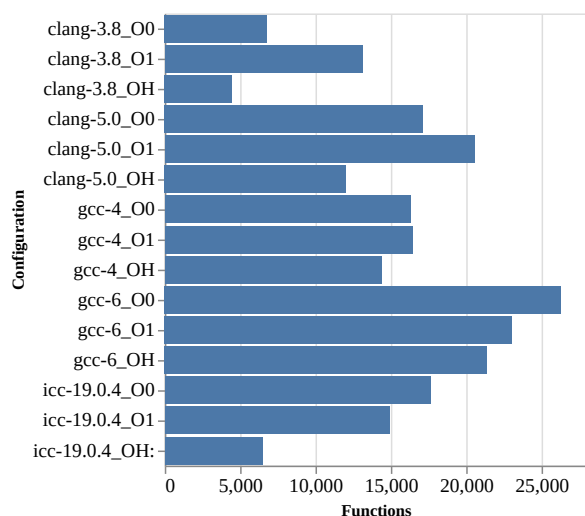


Figure 8: The distribution of samples in compiler configurations in the NeuralCI dataset (prior to re-balancing).

Figure 8 shows the number of functions for each configuration in the original dataset of NeuralCI [20], for 64-bit dynamically linked and unstripped executables. Some configurations, such as Clang 3.8 at `-OH` optimization level, contain significantly less functions than others. This unbalanced dataset could potentially cause bias in evaluation. Therefore, we balanced this dataset by randomly removing functions such that all configurations have the same amount of functions (see Section 6.1).

Dataset	Precision	Recall	F-1
Unbalanced	83.5%	83.5%	83.5%
Balanced	76.6%	76.6%	76.6%

Table 5: Comparison of NeuralCI results using the unbalanced dataset and the balanced dataset

To replicate the evaluation of NeuralCI as accurately as possible, we ran an additional experiment using Neu-

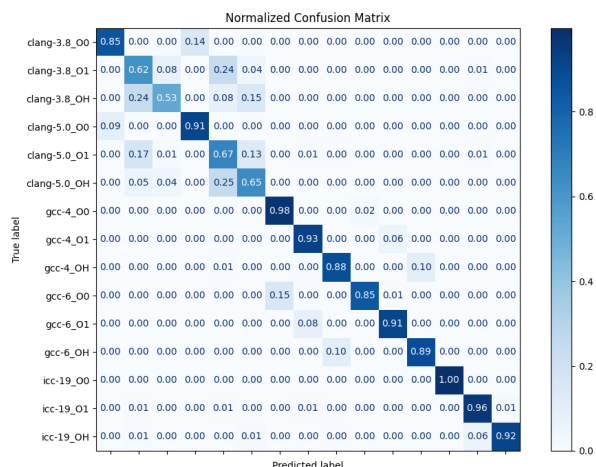


Figure 9: Confusion matrix of NeuralCI using the unbalanced dataset

ralCI with the unbalanced dataset. Table 5 shows the results of NeuralCI using the unbalanced dataset and the dataset after we balanced the data size. The results using the unbalanced dataset show roughly the same results as reported by Tian et al. [20], with a negligible variation (<1%). Notice, however, that after balancing the dataset, NeuralCI’s performance declines. Compared to the confusion matrix of NeuralCI using our balanced dataset (Figure 3), the confusion matrix of NeuralCI using the unbalanced dataset (Figure 8) shows drastically better results for identifying the GCC version at `-OO` optimization level, likely because GCC 6 has significantly more samples than GCC 4 at `-OO` optimization level. Likewise, the results for Clang functions also differ.

B Artifact Appendix

Abstract

Our artifacts include the dataset for our experiment in Section 6.1, 6.2 and 6.4, our coarse-grained compiler configuration classifier for Section 6.1, and our fine-grained compiler pass classifier for Section 6.2 and Section 6.4. Our artifacts require a Linux machine (or Windows Subsystem for Linux) with 32GB of RAM and 16GB of storage. Since our classifiers use only shallow learning, a discrete GPU is not required. On our machine with an AMD Ryzen 7 3700X processor, the coarse-grained classifier requires about two hours to finish, and the fine-grained classifier takes about an hour.

Scope

The artifacts allow reproducing our quantitative experiments in Section 6, including coarse-grained compiler configuration identification (Section 6.1), optimization pass identification using only static features (Section 6.2),

and optimization pass identification using both static and dynamic features (Section 6.4).

Contents

Our artifacts include the following contents:

1. `balanced_dataset.csv`: The dataset for coarse-grained compiler configuration classification. As discussed in Section 6.1, this dataset is a balanced subset of the dataset used in NeuralCI [20].
2. `config_classifier.py`: The coarse-grained compiler configuration classifier.
3. `data.csv`: The dataset for fine-grained compiler pass classification used in Section 6.2.
4. `data_dynamic.csv`: The dataset for dynamic feature evaluation used in Section 6.4. As discussed in Section 6.4, this dataset is a subset of `data.csv` that only includes functions whose dynamic feature coverage are at least 70%.
5. `LICENSE`: The license of our artifacts.
6. `passtell.py`: The fine-grained compiler pass classifier.
7. `README.md`: Installation and running instructions.
8. `requirements.txt`: List of required Python libraries.
9. `static_opcode_features.py`: Library module required for `passtell.py`.

Hosting

The classifiers and the datasets are available at <https://github.com/zeropointdynamics/passtell> in the main branch with commit ID 0c88e8d.

Requirements

Our classifiers have the following requirements:

1. A 64-bit Linux machine with at least 32GB of RAM and 16GB of storage. We have tested our artifacts on Arch Linux (rolling release, updated in May 2022) and Ubuntu 20.04 (Windows Subsystem for Linux).
2. Python 3. For Ubuntu and other Linux distributions that do not have a default `python` command, setting the symbolic link from `python` to `python3` is required. On Ubuntu, this can be done by installing the `python-is-python3` package.
3. Graphviz.
4. Python libraries listed in `requirements.txt`.



JITServer: Disaggregated Caching JIT Compiler for the JVM in the Cloud

Alexey Khrabrov
University of Toronto

Marius Pirvu
IBM

Vijay Sundareshan
IBM

Eyal de Lara
University of Toronto

Abstract

Managed runtimes such as the Java virtual machine (JVM) rely on just-in-time (JIT) compilers to improve application performance by converting bytecodes into optimized machine code. Unfortunately, JIT compilation introduces significant CPU and memory runtime overheads. JIT compiler disaggregation is a technique that decouples the JIT from the JVM and ships compilation to a separate remote process. JIT disaggregation reduces overall memory usage; however, its communication overheads result in higher system-wide CPU usage.

JITServer is a disaggregated caching JIT compiler we implemented in the Eclipse OpenJ9 JVM. It improves system-wide resource utilization by enabling the caching of compiled native code and its reuse in JVMs running on different machines. JITServer is transparent to the application developer, and supports all the dynamic features in the JVM specification. In our experiments, JITServer reduced overall CPU cost by up to 77%, overall memory usage by up to 62%, application start time by up to 58% and warm-up time by up to 87%.

1 Introduction

Java virtual machines (JVMs) rely on just-in-time (JIT) compilers to improve the performance of Java applications by converting the bytecodes of the application into optimized machine code. Since this transformation is done at runtime, the JIT has the ability to tailor-fit the generated code for a specific application instance and its execution environment. On the downside, JIT compilation can introduce significant runtime overheads in terms of processing power and memory. The extra CPU cycles needed for compilation can interfere with applications' progress, delaying their start-up, increasing their warm-up time or affecting the response time and quality of service. Similarly, the data structures allocated by the JIT compiler create unpredictable spikes in memory usage, which increase memory footprint and can lead to performance degradations (due to paging) and out-of-memory failures. In our experiments, JIT compilation accounted for up to 50% of CPU time used during the start-up and warm-up phases, and for up to hundreds of MBs of memory footprint.

The competition for resources between the application and the JIT is more intense in CPU and memory constrained environments such as containers and VMs found in cloud datacenters that maximize resource utilization and application density.

Automatic scaling of cloud applications is done by launching and shutting down instances based on load. Frequent restarts of applications pose serious challenges to Java workloads due to the high start-up overhead of JIT compilation, which needs to be amortized over a long execution period. The memory overhead of JIT compilation is more significant for smaller (in terms of overall memory usage) application instances which are common in the cloud (e.g. microservices).

JIT compiler disaggregation is a technique that addresses these overheads by decoupling the JIT from the JVM and running it in a separate remote process. The JIT no longer steals CPU cycles from the application, which leads to more predictable behavior and better quality of service, and improves application warm-up in CPU constrained environments. Memory footprint spikes are also eliminated, enabling smaller containers, higher application density, and reduced costs in the cloud. Moreover, remote JIT simplifies resource provisioning: the user only has to consider CPU and memory required for application execution, while compilation resources can be scaled independently of the applications.

While JIT disaggregation reduces overall memory usage, on the downside, it can result in higher system-wide CPU usage. The CPU cost and latency of each compilation in this setting is higher compared to local JIT due to communication overheads. JIT overheads are not eliminated, but rather transferred to a different host, at the expense of additional networking and serialization costs.

We argue that in order to achieve the full benefits of disaggregated JIT, the compiler server resources must be effectively shared between multiple client JVMs by making it possible to reuse compilations of common methods. Unfortunately, reusing dynamically compiled native code across multiple JVM instances is a challenging task. JIT-compiled code cannot be simply plugged into a different JVM in the general case since it often contains pointers to runtime entities that are located at different addresses in different JVMs, and relies on runtime assumptions that might not hold in a different JVM environment. Due to the dynamic nature of the JVM, locating runtime entities and verifying assumptions across JVM processes is more difficult compared to relocating statically-compiled code in languages like C. The key idea is to use secure hashes of immutable class metadata to efficiently detect equivalent classes and methods across JVMs.

In this paper, we describe the design and implementation

of JITServer - our disaggregated JIT compiler in Eclipse OpenJ9 [6], a popular open source JVM. JITServer caches compiled native bodies and reuses them for future compilation requests for the same methods from other client JVMs. This enhancement dramatically decreases compilation latency for cache hits and significantly reduces overall resource usage by amortizing compilation costs over multiple clients. Caching compiled code at the server happens transparently and does not add any complexity to application development. Our main use case is running multiple application instances in a cloud datacenter in, e.g., containers with resource limits. Remote JIT might not be beneficial if the JVM has plenty of resources for local JIT, or if the network latency is high.

This paper makes the following contributions:

- We propose a novel mechanism that facilitates caching of compiled native code in a remote JIT compilation system and enables correct, transparent and efficient reuse of such code by JVMs running on different machines. We show that caching is necessary to achieve the full benefits of JIT compiler disaggregation.
- We describe the design and implementation of JITServer. Unlike previous work, our solution is compliant with the JVM specification, does not rely on simplifying assumptions, and is implemented in a production grade JVM with a sophisticated JIT. We provide insight into the challenges of implementing remote JIT in a dynamic environment such as the JVM and the ways to solve them.
- We present the first (to the best of our knowledge) study of remote JIT in the context of cloud computing. We show that JITServer improves start time, warm-up time, CPU and memory usage, without trading-off peak throughput, allowing more efficient, higher density deployments of JVM-based applications in the cloud. In our experiments, JITServer reduced overall CPU cost by up to 77%, overall memory usage by up to 62%, application start time by up to 57% and warm-up time by up to 87%.

The rest of the paper is organized as follows: Section 2 provides a survey of related work and motivates our solution; Section 3 presents the design of JITServer and its novel mechanism for reusing compiled code in multiple JVMs; Section 4 evaluates the performance of our system; finally, Section 5 concludes the paper and explores future work directions.

2 Motivation and Related Work

In this section we present a survey of existing solutions for the JIT overhead problem and discuss their limitations.

2.1 Static AOT Compilation in the JVM

One way to circumvent the negative effects of JIT compilation is to use static ahead-of-time (AOT) compilation. HotSpot JVM used to include a (now deprecated [10]) static AOT compiler `jaotc` [9] that compiled the bytecode of an explicitly specified list of Java classes or `.jar` files into native code.

GraalVM Native Image [7] compiles a Java application including all the classes it uses (determined by static analysis) into a standalone native executable, and can run parts of the application initialization code at AOT compile time [31].

However, an inherent limitation of static AOT compilation is the *closed world assumption*: all the code that can execute at runtime must be available at compile time. This assumption severely limits support for dynamic JVM features such as custom dynamic class loading, class definition and redefinition at runtime, and `invokedynamic` bytecodes. Static AOT only supports a subset of Java and JVM bytecodes.

On the performance front, static AOT compilers typically do not take advantage of the latest CPU features, because the code they produce must be compatible with a wide range of target machines. Moreover, the lack of runtime profiling information can lead to suboptimal performance. While it is possible to use profiling information at build time, it requires a realistic workload, which makes application development more difficult. In addition, performance profile of a given method can change between application phases, and achieving peak performance in such cases still requires dynamic recompilation at runtime.

2.2 Sharing Compiled Code between JVMs

JIT overhead can be reduced by caching and sharing compiled code among JVMs. Examples include ShMVM [20] (based on HotSpot), ShareJIT [32] (based on Android Runtime), and the Shared Classes Cache (SCC) [16, 19] and dynamic AOT compilation in OpenJ9. We focus on the latter as the more recent and practical implementation of this approach.

SCC in OpenJ9 is a memory mapped file used to cache compiled code and the internal representation of immutable class metadata. The SCC is populated in a *cold* run and is subsequently consumed by other JVM instances in *warm* runs. The SCC improves start-up and warm-up performance in the warm runs since the class metadata is already available in a pre-processed format and does not need to be parsed from the class files, and loading cached compiled methods is much less CPU-intensive than JIT-compiling them. This approach is called *dynamic AOT compilation*: methods are compiled and stored in the SCC during execution, in contrast with static AOT where the code is compiled before it runs.

Unfortunately, this approach does not completely eliminate the need for a JIT compiler for two reasons: (i) the hit rate in the SCC is not 100% as the set of compiled methods can vary from run to run; and (ii) dynamic AOT code can be slower than regular JIT-compiled code since it has to meet certain constraints in order to be relocatable and usable in a different JVM instance. Therefore, performance critical methods are still JIT-recompiled with more optimizations in order to achieve peak throughput. Since such compilations are responsible for most of the JIT memory overhead, this approach cannot effectively reduce peak memory usage.

While it is possible to ship a pre-populated SCC with an ap-

plication, the complexities involved often make it impractical. The associated increase in image size can be significant up to hundreds of MBs (63-128% increase for the applications we used in our evaluation). A larger image size adds overhead on the critical path of deployment and contributes to the cold start latency. Dynamic AOT code makes assumptions about the execution environment such as target CPU instruction set, GC algorithm (its reference read and write barriers), and heap size (determines compressed pointer shift). Shipping a pre-populated SCC requires either maintaining multiple versions for all possible combinations of CPU generations and JVM parameters (which complicates deployment), or generating suboptimal portable code that works across all configurations.

Managing the pre-populated SCC puts additional burden on application developers and increases complexity and cost of continuous integration and deployment. Caching methods compiled during warm-up requires simulating a realistic workload, which can be a complex task. Creating a fully populated SCC can also increase application build times by up to orders of magnitude since it can take minutes of application run time to achieve full warm-up. Anecdotal evidence (e.g. Docker images for OpenJ9 [2] and Open Liberty [11] - a popular Java framework optimized for OpenJ9, and the Java runtime in the OpenWhisk [4] serverless platform) suggests that in practice, the SCC is typically pre-populated only by starting up and shutting down an application instance, and does not include any methods compiled during the warm-up phase.

Another way to leverage the SCC is to share it locally between JVMs on a given host, populating it dynamically at runtime instead of pre-populating it at application build time. However, this approach also has drawbacks. Sharing the SCC between applications creates the potential for side-channel attacks and other security issues. Thus if sharing is limited to a single application, the scheduler is forced to pack instances of the same application on the same host, which can lead to “hot spots” during load spikes when multiple instances of the same application contend for (often oversubscribed) resources. This approach also increases applications’ exposure to individual host failures. Managing per-application SCC volumes shared between containers also complicates deployment.

2.3 Checkpointing and Reusing JVM Processes

Another approach to circumvent JVM slow start is to checkpoint the state of a “warm” JVM process and restore it when starting a new application instance. Cloneable JVM [24], ReplayableJVM [30], and Catalyzer [22] explored checkpointing well-defined and deterministic state after the start-up phase of the application. Such snapshots do not include any methods compiled during warm-up under load, and still require JIT compilation to reach peak performance. Checkpointing also suffers from the same usability issues as shipping a pre-populated cache of compiled methods: additional developer effort and having to generate slower portable code or maintain multiple versions for different CPUs and JVM configurations.

HotTub [27] takes a somewhat different approach of reusing JVM processes to keep the JVM state “warm” for the next run of the same or similar application. Photons [23] co-locates multiple copies of a serverless function as threads within the same JVM instance. These systems only reuse compiled code within the same machine, and can only persist it across non-concurrent invocations by keeping the pre-warmed JVMs running, which incurs a significant idle footprint.

2.4 Remote JIT Compilation

Remote JIT has been originally proposed in research that focused on embedded, mobile, and IoT devices where local JIT is prohibitively expensive in terms of memory, CPU or energy consumption [17, 18, 21, 25, 28, 29]. JCOD [21], MoJo [28] and VM* [25] are based on simplified JVMs and JIT compilers that either do not rely on dynamic JVM runtime information, or only support static AOT compilation of a subset of Java.

Lee et al. [26] describe a JIT compilation server based on Jikes RVM, a research JVM written in Java. To the best of our knowledge, their work is the state-of-the-art in the literature on remote JIT for the JVM. Its design assumes that all the information needed to compile a method is included in the compilation request, which becomes impractical in a complex modern JIT. The authors do not consider overall resource usage (including the server), and only use simulations to evaluate performance with multiple clients.

Foremost, these previous approaches do not reduce, and in fact can increase, system-wide CPU usage. Each individual remote compilation consumes more CPU time than its local equivalent (assuming homogeneous hardware) due to communication overheads, and takes more time overall due to network latency. As we show in our evaluation (see Section 4.2), remote JIT can increase overall CPU usage, especially for short-running workloads that are common in modern cloud computing. In this paper, we leverage caching compiled methods at the server to reduce the overall CPU cost by amortizing it over multiple clients. In addition, we show that for cloud workloads, JITServer significantly reduces overall memory consumption since the spikes of maximum memory usage from multiple clients are unlikely to align.

Azul Cloud Native Compiler [5] is a recently released remote JIT for the Azul JVM. Unfortunately, there is only limited technical information available about the design of this proprietary closed-source system. In particular, it is not known if it implements caching, or how it affects system-wide resource usage. We are unable to compare performance against it since its license forbids publishing benchmarking results.

Remote JIT is less susceptible to failures than other disaggregated designs (e.g. memory disaggregation) since it has no shared hard state. Unlike the case of mobile and IoT devices (the main target of previous remote JIT work), network latencies in cloud datacenters are relatively low, and our evaluation shows that remote JIT performs very well in this setting. Modern cloud applications themselves are also typically dis-

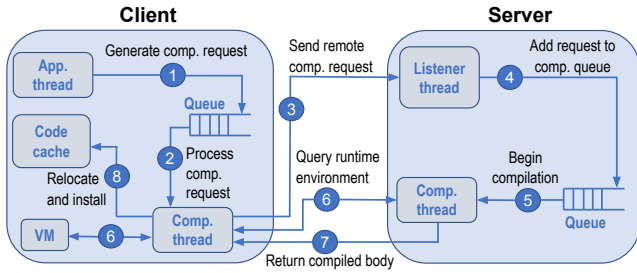


Figure 1: Remote compilation mechanism

tributed, thus remote JIT does not exacerbate the reliability and latency concerns. Remote JIT shifts resource provisioning complexity from the application to the infrastructure, which can be arguably beneficial. Local JIT requires application developers to manage the complexity and extra costs of over-provisioning memory (which goes unused after warm-up) and CPU (to maintain QoS despite JIT activity during warm-up) for each JVM. Instead, the operator’s effort to setup JITServer autoscaling can be reused many times across applications.

3 Design and Implementation

In this section we present an overview of JITServer design and implementation. We start with the description of the remote compilation mechanism, and then explain how we enable caching compiled methods to be reused by multiple clients.

3.1 Remote Compilation Mechanism

Figure 1 shows the high-level design of remote compilation in JITServer. The *client* on the left is a complete JVM running an application, while the *server* on the right is a separate, possibly remote process offering a JIT compilation service shared by multiple client JVMs connected to it. In this paper, we focus on the mechanisms. We plan to explore automatic sizing and scaling of JITServer resources in future work.

Existing remote JIT designs assume that each compilation is a simple request-reply operation that requires a single network round-trip since all the data required by the server is either already available there or included with the request [26]. While this was feasible for simple JIT compilers in previous work, such a design proved difficult to implement in a modern JVM. The JIT in OpenJ9 has over 100 optimization passes that use over 100 types of JVM state queries, compared to ~40 optimizations and 12 types of state in [26]. We took a different approach that resulted in lower implementation effort and complexity: a compilation request only carries data that is likely to be used by most compilations, and remaining data is requested from the client on demand, cached at the server, and invalidated when necessary to ensure correctness.

OpenJ9 runs a number of background threads that perform JIT compilation concurrently with application threads. It uses *tiered* compilation: methods are scheduled for compilation at lower optimization levels once they reach certain invoca-

tion thresholds, and particularly hot methods that consume a significant portion of CPU time are scheduled for recompilation at higher optimization levels. The bytecode interpreter and the sampling thread add compilation requests to a queue consumed by the compilation threads. The number of active threads is adjusted dynamically: additional ones are activated when the JIT is starved of CPU time (e.g. if there is a large number of application threads) or the queue becomes too large. Compilation threads compile one method at a time, and each compilation is a single-threaded task.

JITServer uses the same basic design with compilation queues and threads on both the client and the server. At Step 1, application threads add requests to the compilation queue. At Step 2, a client compilation thread dequeues a request and sends a remote compilation request to the server (Step 3). The client sends enough data for the server to start the compilation, such as the bytecodes of the method. A listener thread on the server accepts the connection and enqueues a compilation request (Step 4), and a compilation thread reads the request data from the network and starts the compilation (Step 5).

During remote compilation, the server may issue various queries to the client requesting information about classes, methods, fields, execution environment, etc. (Step 6). When the compilation is complete, the server sends the compiled body to the client (Step 7) along with metadata that needs to be instantiated at the client. The client performs necessary relocations (e.g. fixes up calls to runtime helper methods) and installs the compiled method in its own code cache (Step 8) - now it is ready to be executed. As long as the compilation queue is not empty, the client thread keeps the connection open and reuses it for subsequent requests.

Each compilation is performed by a dedicated pair of threads - one on the server and one on the client. The number of in-progress compilations is limited by the maximum number of client threads (16 in the current implementation). Our evaluation shows that this value works well in practice. We used a maximum of 128 server threads in our experiments.

The server JIT compiler in our current implementation is identical to the local OpenJ9 JIT compiler and uses the same set of optimizations and heuristics that control its behaviour. All compilations are performed remotely. We focus on evaluating the benefits of JIT disaggregation and caching on their own in this paper, and plan to explore how the compiler can be improved in the disaggregated setting in our future work.

3.2 Caching JVM Runtime Information

Remote compilation of large or complex methods can require exchanging a large number of messages between the client and the server. To reduce the negative effect of network latency on performance, the server aggressively caches runtime information received from the clients. Note that this is different from caching the resulting compiled code, which we describe in Section 3.4. Caching is essential for performance: in an early JITServer prototype that did not implement such

caching, client threads alone used ~10x more CPU time than local JIT. Caching reduced the average number of messages per compilation from over 1000 to ~40.

The server handles requests from multiple clients in parallel and maintains their data in *client sessions* identified by a unique client ID included with each compilation request. A session is created when a client first connects to the server, and destroyed when the client terminates or stays inactive for a long time. Most of the information that does not change throughout the client's lifetime (e.g. JVM configuration and class metadata for primitive types) is sent with the first compilation request. Mutable client state needs to be synchronized with the server as classes are being loaded, unloaded, or redefined, and profiling information is updated. We describe how we handle caching of important types of client data.

Class Metadata Java classes are represented in OpenJ9 as data structures called ROMClasses, which represent immutable data, including method bytecodes, and RAMClasses, which contain the mutable data and point to the ROMClasses they are based on. Multiple RAMClasses can use the same ROMClass, e.g. if they are loaded by different class loaders.

The server maintains mirrors of the clients' ROMClasses and RAMClasses. To reduce memory consumption, it stores a single copy of each unique ROMClass that is shared by all the clients using it. To handle class unloading and redefinition, the client sends with each compilation request the list of classes unloaded or redefined since the previous compilation request, which are in turn deleted from the server cache.

Server compilation threads are synchronized with class unloading and redefinition using a reader-writer lock. Threads acquire this lock for writing before deleting the entries, and acquire it for reading while performing a compilation. When runtime information queries are sent to the client, the lock is released, and the client's reply will also indicate whether class unloading happened during this compilation, in which case the compilation is aborted and retried.

Class Hierarchy Table CHTable (class hierarchy table) is an OpenJ9 data structure that captures relationships between classes and enables fast discovery of such relationships used for speculative optimizations, e.g. devirtualization. The server mirrors the client's CHTable and keeps it up to date with incremental updates sent along with the compilation requests.

To guarantee functional correctness, the server must process class unloading/redefinition and CHTable updates in the same order as the client. This requirement imposes a partial order on compilation requests: *critical* requests that carry updates must be processed in the same order as they originated at the client, while other *non-critical* requests between subsequent critical requests can be reordered. We use a sequencing scheme to order critical requests: if the server receives an update with an out-of-order sequence number, it suspends the compilation thread until the update with the expected sequence number arrives. If the expected message is lost, after a timeout, the server clears cached data for the client and restarts

the update mechanism by requesting the entire CHTable.

Profiling Data While interpreting Java methods, OpenJ9 collects profiling data about branch direction and targets of virtual and interface calls and `instanceof/checkcast` operations. The optimizer at the server makes extensive use of this data and issues many queries to the client. To reduce the number of messages, we batch profiling data requests by sending the data for all the bytecodes in the method in a single reply.

Unlike class hierarchy information, imprecise profiling data does not affect functional correctness of generated code. Profiling data for already compiled methods (which can be inlined into newly compiled ones) has stabilized and can be cached without significant effect on performance. In contrast, profiling data for interpreted methods continue to be accumulated, and using a stale version can lead to bad optimization decisions. Thus, the server caches this data only for the duration of the current compilation.

3.3 Reliability and Security

The client JVM in our design still includes a fully functional local JIT compiler. Out-of-process compilation makes Java applications more resilient to JVM crashes caused by intermittent software bugs in the JIT compiler. JITServer only maintains soft state, allowing transparent failure handling: after a server crash, the client JVM can switch to a different JITServer instance or compile the method locally (which is unlikely to trigger the same bug again) and continue execution, and the failed JITServer instance can be simply restarted.

We assume a security model where the server instance and all the client JVMs that connect to it are in the same security domain: the clients need to trust the server to generate correct native code. JITServer supports encrypted communication using OpenSSL. Encryption adds a relatively small but non-negligible overhead: it increases start time by up to 9%, warm-up time (to 90% of peak throughput) by up to 5%, and CPU time used by the JIT by up to 5%. Unencrypted communication can be used in settings with an isolated network (e.g. on-premises cloud deployments) to reduce the overhead.

Even if communication is encrypted, remote JIT can potentially introduce side channels for an in-network attacker that can observe the timing and sizes of messages exchanged between the server and the client JVMs. However, the same issue applies to any distributed or client-server application, and in practice does not prevent their widespread adoption.

3.4 Reusing Dynamically Compiled Code

OpenJ9 is capable of producing relocatable code (aka dynamic AOT code) that can be reused in a different JVM instance running on the same machine. It maintains a Shared Classes Cache (SCC) - a memory-mapped file shared by JVMs on the same host that stores immutable class metadata and dynamic AOT code. These cached methods include additional metadata that is used to locate and patch pointers to JVM runtime entities such as RAMClasses (see Section 3.2)

residing at different addresses in different JVMs, and to verify that assumptions (e.g. about class hierarchies) made during compilation still hold in a different JVM environment.

Dynamic AOT methods refer to SCC entities in their *validation and relocation records* by offset within the SCC. While this approach is very efficient when running on a single machine, this means that dynamic AOT code cannot be reused as-is on a different machine (with a different SCC). Since the order of class loading and dynamic AOT compilations is not deterministic, SCC entities will generally reside at different SCC offsets on another machine, making dynamic AOT code generated on another machine invalid. The fundamental source of this problem is the tight coupling of the dynamic AOT code with the SCC it is stored in. We propose a new scheme to store these artifacts independently.

Our caching mechanism is based on the existing OpenJ9 infrastructure for relocating dynamically compiled code, but uses a different way of identifying runtime entities such as classes across JVMs. Our novel scheme allows us to decouple compiled code from class metadata and enable its reuse in any JVM running on any host. We store compiled methods in a *serialized* format that refers to JVM runtime entities by globally unique identifiers instead of SCC offsets. When another client JVM requests a compilation of the same method, the server replies with a cached serialized version. The client then *deserializes* the method by finding the corresponding runtime entities, and then relocates and loads the native code.

Some compilations (e.g. hot methods at higher optimization levels) do not use the relocatable format in order to maximize the performance of generated code, since relocation limits optimization opportunities. The resulting hit rate in the JIT-Server cache is 85-93% during the start phase and 68-76% during warm-up in our experiments, depending on the application. In our future work, we will explore compiling more methods as relocatable code to increase the cache hit rate.

The performance of JIT-compiled code is affected by the quality of profiling data, leading to a trade-off between compiled code reuse and possible performance degradation due to differences in profiles across the clients. We focus on the common use case of sharing between instances of the same application where profiles are likely similar (e.g. horizontal autoscaling, FaaS, data-parallel computation). In other cases, we expect the effect to be limited since particularly hot method compilations most affected by conflicting profiles are not cached and rely on individual clients' profiles. We plan to investigate the effect of profiling data variability on the performance of reused native code in our future work.

3.5 Method Serialization Mechanism

The main building block of dynamic AOT relocations is identifying equivalent classes and methods across JVMs. Examples include *inlining guards* (checking an object's class before executing the inlined body) and calls to other compiled methods. Such instruction sequences contain addresses of `RAMClasses`

and methods that need to be patched in a different JVM.

We add another level of indirection to relocatable code by effectively serializing relocation and validation records. For each runtime entity they refer to, the server stores a corresponding *serialization record* with enough information for a client JVM to find the entity and verify that it is equivalent to the one used in the original compilation. Each serialization record also contains the offset from the start of the AOT method body to the location of the corresponding SCC offset field stored in the validation or relocation record.

When the server performs a dynamic AOT compilation in response to a client request, it serializes the compiled method and stores it in its in-memory cache. Serialized methods are self-contained and can be persisted to disk and shared across multiple JITServer instances. In our future work, this will enable efficient autoscaling of JITServer resources by launching new instances with a warm cache persisted from an existing instance. When a different client JVM receives a serialized AOT method, it iterates through the serialization records, looking up the corresponding entities and updating the SCC offsets to them with their local versions. After successful deserialization, the client proceeds to store the resulting method body in its local SCC (so that execution environments that do not get torn down can take advantage of it in the next run), and loads the method as regular dynamic AOT code.

If any lookups or validity checks fail, deserialization is aborted and the client JVM requests a regular non-cached compilation from the server. Deserialization failures occur infrequently during normal operation: the failure rate for the applications we used in our evaluation is 1-5% during the start phase and less than 1% during warm-up. Such failures happen because the set of classes loaded by the time a method is compiled varies from run to run, therefore a small number of lookups fail as the classes have not yet been loaded.

Relocation and validation records can refer to the following SCC entities: *ROMClass* - immutable part of class metadata; *ROMMethod* - immutable part of method metadata including its bytecodes (part of `ROMClass`); *class chain* - a list of `ROMClass` SCC offsets for classes and interfaces a given class extends and implements.

Identifying Classes and Methods We identify a class across JVMs using a combination of its fully-qualified name and a secure hash (e.g. SHA-256) of the `ROMClass`. We use the hash to efficiently check that a client JVM's version of the class is the same as the one used during compilation. Since the `ROMClass` contains the full description of the class including the bytecodes of all methods, a matching hash guarantees correctness. We identify methods by their defining class, name, and signature (types of parameters and return value).

We use class chains to verify that the whole inheritance chain of a given class is the same across two JVMs. For example, if one of the superclasses or interfaces of a given class is redefined at runtime in the JVM that loads the method, the class chain will not match, even though the class itself has

not changed. The serialization record for a class chain is a list of class serialization records for each class in the chain.

Identifying Class Loaders Since Java classes can be loaded by application-defined class loaders, a class lookup in a running JVM requires a class loader (in addition to class name). We use the following heuristic for class loader identification: we associate each class with the identity of the first class loaded by its class loader. While a regular AOT method refers to a class loader by the class chain of the 1st class that it loaded, a serialized AOT method refers to it by the *name* of the first class that it loaded. We maintain the 3-way mapping between a class loader and the name and the class chain of the first class that it loaded, in each client JVM. We update this mapping at runtime when the JVM loads or unloads classes and creates or destroys class loaders.

The identification heuristic can fail in edge cases described below, however, that does not affect the correctness of compiled code. While we observed no class loader identification failures in our evaluation, they are still possible in rare cases.

Assume that in the compilation environment, RAMClass C1 is the first one loaded by L1. Cached methods that refer to L1 identify it by the name of C - the ROMClass that C1 is based on. In the load environment, RAMClass C2' is the first one loaded by L2' - a different class loader, and then later RAMClass C1' is the first one loaded by L1' (the correct class loader matching the compilation environment) from the same ROMClass C. As a result, loading a cached method can result in using a wrong RAMClass (one loaded by the incorrectly guessed L2'). However, RAMClass pointers are only used directly in generated code in *guards* as described above. A mismatching RAMClass pointer can only affect performance (execution will take the slow path of making a virtual call) while correctness is preserved. In other types of relocations, RAMClass addresses are not present in the native code, and are only used to verify class chains and locate RAMMethods, which are shared by both RAMClasses. Another possibility is that L1' might load a different class first, and in this case cached methods that refer to it simply will not be loaded.

JVM Environment Compatibility Each AOT method is implicitly associated with an *AOT header* - a data structure that describes the compilation environment: CPU features, JVM configuration, etc. Local SCC stores a single instance of this structure. All JVM instances that store or load AOT code in the SCC must have a matching configuration. Serialized AOT methods store the AOT header of the JVM that it was originally compiled for. To serve a client compilation request from the cache, the server looks up a serialized method with a compatible AOT header.

Storage and Transfer Optimizations In order to make the serialized AOT method representation more compact for optimal storage and transfer, we store serialization records at the server separately from method bodies, and serialized AOT methods refer to them by unique IDs. When responding to

a client compilation request with a cached serialized AOT method, the server sends the serialized method body along with all the serialization records it refers to that the client has not yet received. In order to reduce network traffic and deserialization overhead, the client caches serialization records received from the server, and the server keeps track of the record IDs that are already cached at the client.

The size of a populated JITServer cache of compiled methods is ~30-130 MB in our experiments depending on the application, which is smaller than the pre-populated local SCC (~65-170MB) since the JITServer cache only stores ROMClass hashes instead of full class metadata. Since the set of compiled method varies across multiple clients (even running the same application), the JITServer cache accumulates a larger number of methods compared to the local SCC which is only populated once. The cache size can be reduced by pruning the "tail" of less popular methods, e.g. ones that do not get reused within a certain time period. In our experiments, 7-12% of cached methods were never reused.

End-to-End Example We provide a simple example that illustrates how a compiled method body is serialized by the JITServer and later deserialized and loaded by a client JVM. Consider the following Java code:

```
abstract class A {
    abstract void m1();
}
class B extends A {
    void m1() { ... }
}
class C {
    static void m2(A o) {
        o.m1(); // inlined
    } // as B.m1()
}
```

Assume that the JIT has inlined the call to `o.m1()` in `C.m2()` as a devirtualized call to `B.m1()` as profiling showed that the runtime class of `o` is normally `B`. The inlined body of `B.m1()` (see pseudo-assembly below) is preceded by a *guard* that checks that the RAMClass of `o` is indeed `B`, otherwise it jumps to the slow path that makes a virtual call.

```
cmp rax, ramclass_B; rax contains RAMClass of o
jne slow_path      ; ramclass_B is hard-coded
...                ; inlined body of B.m1()
slow_path: ...     ; virtual call to o.m1()
```

Figure 2 illustrates the entities described below. The metadata of the dynamically AOT-compiled method `C.m2()` contains a relocation record for class `B` that will be used to patch the RAMClass address in the comparison instruction in the guard when the method is loaded in another JVM. This record stores the SCC offset of the class chain for `B`, and the SCC offset of the class chain identifying its class loader `L`. This identifying class chain is the one for the first class that was loaded by `L`. Assume that this class was `Object`, i.e. `L` is the bootstrap class loader. The class chain for `B` is a list of SCC offsets of ROMClasses `B`, `A`, `Object`. The class chain identifying `L` is a single SCC offset of ROMClass `Object`. The SCC offsets are only valid for the client JVM that originally requested this compilation.

To serialize `C.m()`, the server creates the following records corresponding to the relocation record for class `B`:

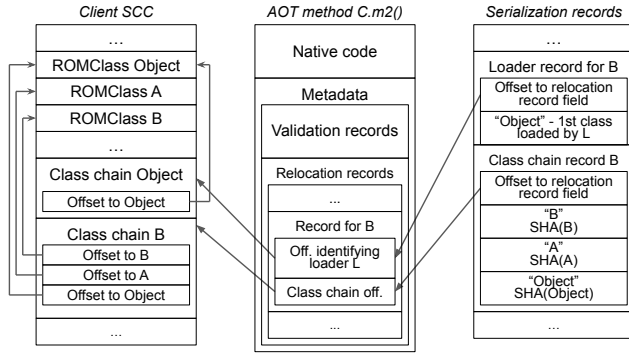


Figure 2: Serialization records for method `C.m2()`

- Class loader serialization record identifying L by the name of the first class it loaded - "Object".
- Class chain serialization record for B .
- Class serialization records for B , A , $Object$. Each record contains the hash of the ROMClass and its name.

When a different client JVM receives this serialized method, it performs deserialization as follows:

1. Find the class loader L' as the one that had a class with the name "Object" as the first class that it loaded. In this case it will be the bootstrap class loader.
2. Lookup RAMClass B' by name "B" in the class loader L' .
3. For each class in the class chain (i.e. B' , A' , $Object$), compute the hash of the ROMClass and compare it with the hash stored in the corresponding serialization record.
4. Update the values of the SCC offset fields in the validation record. The class chain offset will now point to the class chain for B' stored in the local SCC, and the loader identifying offset will point to the class chain for $Object$.

Assuming no failures, the method is now deserialized - all its validation and relocation records point to valid SCC entities - and can be stored in the local SCC for future reuse. The client then relocates the method body by patching the RAMClass address in the comparison instruction in the inlining guard so that it points to the RAMClass B' , and installs it in the JVM code cache. The compiled native code of `C.m2()` can now be executed correctly in this JVM.

4 Evaluation

Our evaluation answers the following research questions:

- Is remote JIT compilation efficient without caching?
- How does remote JIT compilation affect start-up and warm-up times and memory footprint of application instances with different CPU and memory constraints?
- What is the effect of caching on the *overall cluster-wide* resource usage and application density?
- Can remote JIT improve performance if the client JVM already has a pre-populated local SCC (see Section 2.2)?
- What is the effect of remote JIT and caching on the latencies of compilation requests, compared to local JIT?

Name	Framework	Database	Methods compiled	
			Start	Total
AcmeAir	Open Liberty	MongoDB	~3,000	~11,000
DayTrader		DB2	~5,000	~25,000
PetClinic	Spring	H2 (in-memory)	~5,000	~6,000

Table 1: Application benchmarks

Type	CPU	Memory	Storage
A	16-core AMD EPYC 7302P	256 GB	2× NVMe RAID0
B	14-core Intel Xeon E5-2680	128 GB	SSD

Table 2: Hardware configuration

- How does caching affect the scalability of JITServer (i.e. how many clients can it effectively serve at the same time)?
- Does caching allow JITServer to handle higher latency?

Applications We used the following 3 applications in our experiments: AcmeAir [1] - an airline booking system, DayTrader [8] - a stock trading platform, and PetClinic [15] - an animal hospital information system (the de-facto main benchmark for the popular Spring framework). All 3 are web applications; information about them is summarized in Table 1. We used Apache JMeter to generate the workload for all the applications. These applications are multi-tier, end-to-end benchmarks that are more representative of cloud workloads than benchmarks like SPECjvm2008 [14], SPECjbb2015 [12], and SPECjEnterprise2018 [13] typically used for JVM performance evaluation. Individual benchmarks in the SPECjvm suite are essentially microbenchmarks from the JIT perspective with a small number of JIT-compiled methods. SPECjbb and SPECjEnterprise are heavy, long-running (over 2 hours) workloads unsuitable for analyzing cold start performance. Moreover, DayTrader is a comprehensive JavaEE benchmark that uses most of the same technologies as SPECjEnterprise.

Experimental Setup We ran the experiments on a cluster of 11 machines as described in Table 2: 8 machines of type A and 3 slightly less powerful machines of type B. All machines run Ubuntu 18.04 and are connected with a 10 GBit/s Ethernet network (used in all experiments unless specified otherwise) and a 100 GBit/s Infiniband network.

Type A machines run the instances of the application, the database, and the JITServer, while type B machines run JMeter instances (one per application instance) that generate the load. We use a single JITServer instance running on a dedicated machine in all experiments. While co-locating JITServer instances with application JVMs is a viable deployment option, we evaluate JITServer in the fully remote setting to show "worst case" performance. Application instances run in Docker containers with 1 CPU and 1 GB of memory, unless specified otherwise. This container size is roughly equivalent to an AWS EC2 t2.micro instance which is commonly used for modern cloud workloads [3]. The number of JMeter threads is chosen to saturate the throughput of the application instance. The number of database instances is chosen for each benchmark such that the DB is not a bottleneck. The JVMs are configured to use the default heap size and GC policy. Each

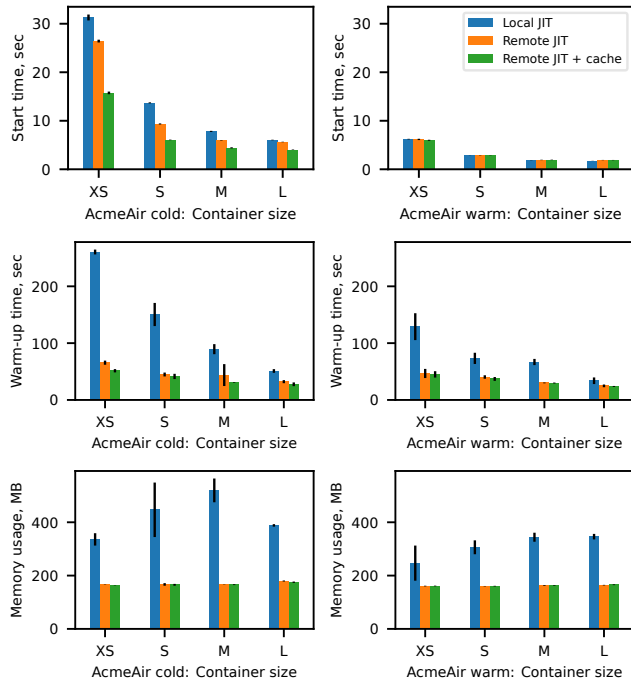


Figure 3: AcmeAir performance: (a) cold; (b) warm

data point is averaged over 5 runs, unless specified otherwise; the error bars on the graphs represent standard deviation.

We compare application performance in three JIT compilation modes: *local*; *remote* without caching, and *remote with caching*. Depending on the experiment, we deploy application instances without a pre-populated local SCC (*cold runs* - default unless specified otherwise), or with an SCC populated by only starting an application instance (*warm-start runs*), or by also applying load to it (*warm runs*). While JITServer supports sharing the cache of compiled code across applications, in this paper we focus on the very common use case of multiple instances of the same application. We do not compare with static AOT compilation in HotSpot and GraalVM. Such direct comparison would not be fair since OpenJ9 is a different JVM, and static AOT only supports a subset of Java. We are unable to compare against the remote JIT compiler in [26] since its implementation has not been made available.

4.1 Application Performance and Footprint

We measure how remote JIT compilation (with and without caching) affects the start-up and warm-up performance and memory footprint of a JVM instance. We run the applications in Docker containers of the following sizes: *XS* (0.5 CPU, 512 MB of memory); *S* (1 CPU, 1 GB); *M* (2 CPUs, 2 GB); and *L* (4 CPUs, 4 GB). When JITServer cache is enabled, it is populated by a single run of the application. We define the performance metrics as follows:

- *Memory usage* - peak resident set size (RSS) of the JVM.
- *Start time* - time since the start of the JVM process until the application is ready to handle client requests.

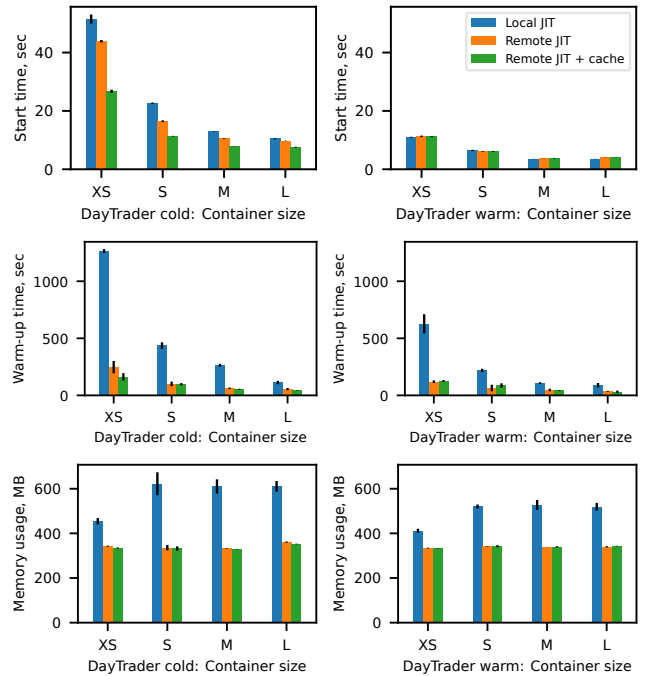


Figure 4: DayTrader performance: (a) cold; (b) warm

- *Warm-up time* - time from the moment the load is applied until the application reaches 90% of its peak throughput.

Figures 3(a), 4(a), and 5(a) show results for *cold* runs of AcmeAir, DayTrader, and PetClinic respectively. Remote JIT without caching reduces memory footprint by up to 68%, start time by up to 40%, and warm-up time by up to 80% compared to local JIT. The addition of caching reduces start and warm-up times even further (especially for smaller container sizes since they have less CPU available for local JIT): by up to 58% and 87% for start and warm-up respectively. There is still a non-negligible number of compilations (including heavy recompilations at high optimization levels) during warm-up that are not served from the JITServer cache: 24-32% depending on the application, compared to only 7-15% during the start phase. As a result, the effect of caching on warm-up time is relatively smaller compared to start time.

Figures 3(b), 4(b), and 5(b) present results for *warm* runs where the local SCC is pre-populated by a previous full run of the application. JITServer still significantly reduces warm-up time (up to 79%) and peak memory footprint (up to 61%), but has no effect on start time since almost all methods compiled during start-up are stored in the SCC and thus do not need to be compiled in a warm run.

Remote JIT with caching is more effective than only using a pre-populated SCC for reducing memory usage and warm-up time. Moreover, these improvements come "for free" as JITServer caching is transparent for the application developer, unlike using the SCC (see section 2.2). Besides, these results represent the advanced and most optimal way to use the SCC which requires very significant developer effort and is rarely

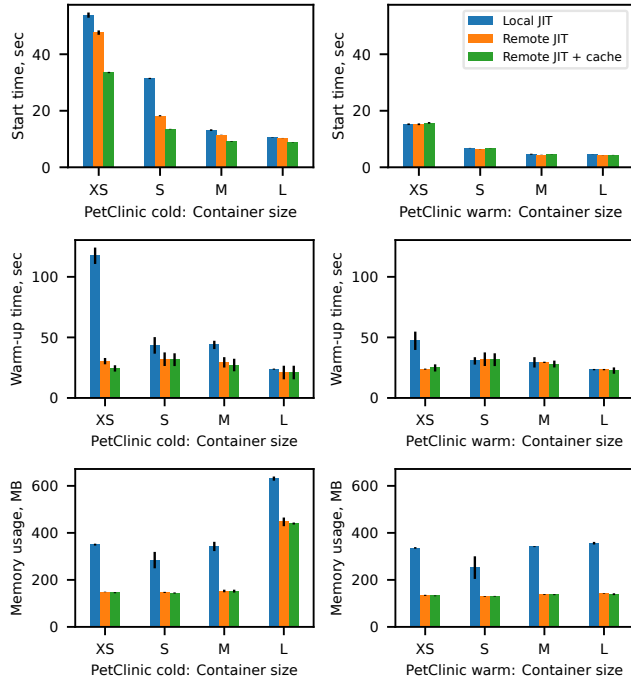


Figure 5: PetClinic performance: (a) cold; (b) warm

done in practice. Caching (local or remote) does not eliminate the need for a JIT compiler. Remote JIT compilation is still a win even with a pre-populated SCC as not all compilations can be cached, and JITServer can make better use of resources, reducing CPU contention and memory footprint.

Application instances reach equivalent peak throughput with local and remote JIT in all the experiments, which is expected since JITServer uses the same set of optimizations and heuristics as the local JIT compiler. The reduction in start and warm-up times is due to the higher degree of parallelism of JIT compilations since the server runs on an additional machine. Arguably, these results do not represent a fair comparison to local JIT as the extra CPU and memory resources used by the JITServer could be used to run other application JVMs instead. In the next subsection we consider *overall system-wide* resource usage, including the JITServer itself.

4.2 Overall System Efficiency

We evaluate the effect of remote JIT compilation (with and without caching) on the *system-wide* resource usage by emulating a cloud deployment where many application instances are brought up and down over a 1 hour period. 64 application slots (one per CPU) are spread evenly across 4 machines. Each slot is used to run a sequence of application instances that execute for a fixed duration (2, 5, or 10 minutes) and stop to be replaced by the next instance. The starting moments of the sequences are staggered with a 10 second interval. We run a single JITServer instance on a separate machine in remote JIT experiments. Each experiment is repeated 3 times.

Figure 6 shows total CPU cost and memory usage (*lower*

is better) in *cold* runs. We define *CPU cost* (measured in milliseconds per request) as the amount of total CPU time used by all JVM instances and the JITServer, divided by the number of JMeter requests served (i.e. useful work done) by all application instances. We use *total CPU time* instead of e.g. aggregate throughput to account for the fact that the JITServer runs on an additional machine that could be used to run more application instances. *Total memory usage* includes the peak RSS of the the JITServer and all concurrent JVM instances.

Remote compilation without caching results in up to 21% *increase* in CPU cost compared to local JIT. On the other hand, caching compiled code at the server effectively amortizes the CPU cost over many clients and results in up to 77% reduction compared to local JIT. The effect is larger for shorter application lifespans since the start-up and warm-up phases with high JIT activity take a bigger portion of the run time. We expect application run times in the cloud to be on the lower end. The CPU cost improvements for PetClinic are relatively smaller since it has fewer methods compiled during warm-up (see Table 1). Remote JIT with caching also significantly reduced start times in these experiments - by 32-58%.

Remote JIT delivers benefits in terms of memory footprint: it reduces total memory usage by up to 62% compared to local JIT compilation. Peak total memory footprint is smaller with JITServer because memory usage spikes caused by heavy compilations from multiple clients are unlikely to align at the server. The 2-minute DayTrader runs are the exception; the extra memory footprint is due to faster heap expansion caused by higher allocation rates since the application reaches higher throughput compared to other JIT compilation modes.

The results for *warm-start* runs (with a pre-populated SCC) are shown in Figure 7. Remote JIT without caching can still have higher CPU cost (e.g. by 9% for 2-minute PetClinic runs) than local JIT with SCC, while JITServer with caching matches or surpasses the performance of local JIT with SCC. Remote JIT still achieves lower (by up to 45%) total memory usage. JITServer not only achieves better resource utilization than using a pre-populated local SCC, but does it transparently without the additional developer effort associated with managing the SCC (see Section 2.2). The two approaches can be combined, in which case adding JITServer reduces CPU cost by up to 53% compared to local JIT.

The main implication of these results for cloud computing is that remote JIT with caching allows to increase application density, even after accounting for the resources used by the JITServer. We can fit more application instances into the same amount of hardware resources since each instance requires less memory and uses the CPU cycles to do more useful computation. With caching, JITServer does not simply move the overhead around - it uses the resources more efficiently.

4.3 Compilation Request Latencies

We measure two compilation latency metrics: (i) *compilation time* taken to serve the request either locally or remotely, and

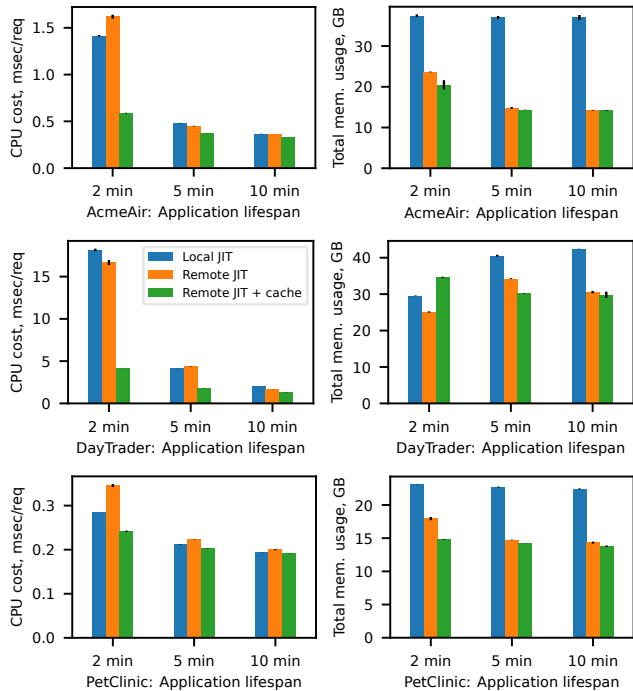


Figure 6: Overall efficiency in cold runs (*lower is better*)

(ii) *total queuing time* - from the moment the compilation is first scheduled by the JVM (e.g. the method reached its invocation threshold) until its completion. Figure 8 shows the resulting CDF (using a logarithmic scale on the X axis) for the AcmeAir benchmark when the JITServer is given all the resources of its host. Figure 9 represents the configuration with *equal CPU resources*: 2 CPUs for local JIT, and 1 CPU each for the JITServer and the client JVM. The results for other benchmarks are very similar and thus omitted.

The results show that individual remote compilations take longer than the local ones, unless the JITServer has more CPU resources. In the latter case, longer (more CPU-intensive) compilations take less time at the JITServer thanks to ample CPU resources, while cheaper compilations take longer than locally - their latency is dominated by communication. Total queuing times are still shorter than with local JIT even with limited JITServer CPU due to increased parallelism: the CPU work is overlaid with waiting for the network. Caching compiled code at the JITServer dramatically reduces compilation request latencies, thanks to the 68-76% cache hit rate.

4.4 Scalability

In order to determine how caching affects JITServer performance with an increasing number of clients, we run a variable number - between 1 and 64 (80 for PetClinic) - of application instances that use the same JITServer instance. Application JVMs start simultaneously and without a pre-populated local SCC, in order to maximize the load on the server. We measure the *full warm-up time* (sum of start and warm-up times) averaged over all concurrent JVM instances. Remote JIT is

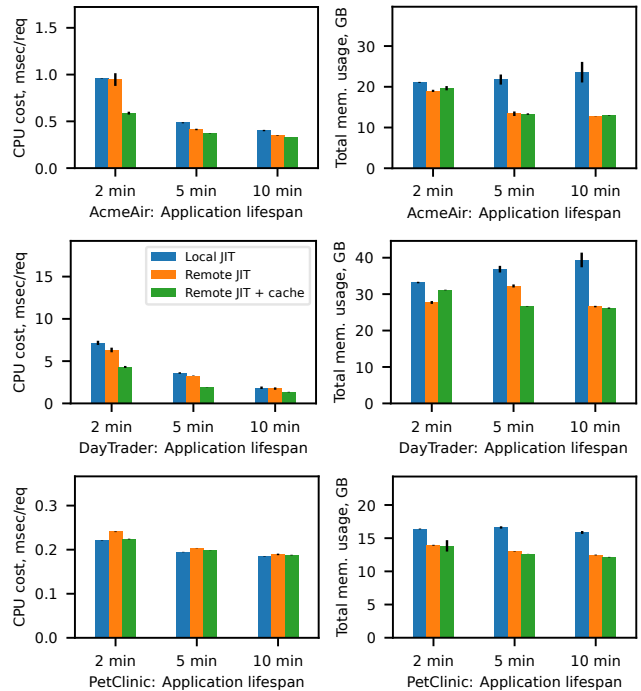


Figure 7: Overall efficiency in warm-start runs (*lower is better*)

more efficient than local for a given number of clients only if the full warm-up time is lower. When the server becomes overloaded, the clients accumulate a backlog of compilation requests, which results in slower warm-up.

Figure 10 shows the full warm-up times for each number of instances (normalized to the local JIT configuration). We see that JITServer caching dramatically improves scalability: given the same amount of resources available to the server instance, it can sustain load from a larger number of clients, while improving their start-up and warm-up performance. The cost of JIT compilation is reduced and effectively amortized over a large number of concurrent clients by avoiding the majority of repeated compilations with caching.

These results represent JITServer performance in a worst case scenario. In a real deployment, we expect client JVM starts to be staggered, providing more opportunities for statistical multiplexing, which in turn should allow JITServer to handle an even larger number of clients without saturating. Additional JITServer instances can be brought up on demand, allowing linear horizontal scaling since JITServer only maintains soft state - the cache of compiled methods.

4.5 Effect of Network Latency

To determine how caching affects JITServer performance with increasing network latency, we run a single application instance with varying values of round-trip network latency between the client JVM and the JITServer. We measure the *full warm-up time* (start + warm-up) and compare it with using local JIT compilation: remote compilation improves performance if it results in faster warm-up. As latency grows,

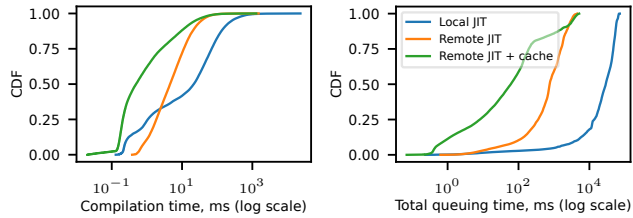


Figure 8: Compilation latencies: unlimited JITServer resources

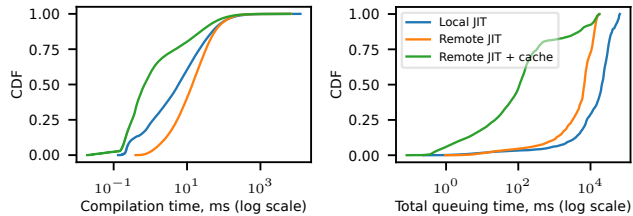


Figure 9: Compilation latencies: equal JIT CPU resources

each compilation request takes more time, and the server eventually becomes unable to compile methods faster than the local JIT compiler. We used the 100 Gbit/s Infiniband network for the smallest latency value - 15 microseconds. The 10 Gbit/s Ethernet network has a latency of 45 microseconds. We emulate the additional latency for subsequent data points using the `netem` module in the Linux kernel.

Figure 11 shows the results for latencies up to over 8 milliseconds. We can see that caching allows JITServer to tolerate higher network latencies (~4-8 ms, compared to ~2-4 ms without caching) since cache hits only incur a single round-trip. Caching compiled code reduces the average number of messages per compiled method by ~54% (from ~42-47 to ~19-22 depending on the application), and the total amount of data transferred per client by ~30% (from ~190-800 MB to ~140-580 MB depending on the application). On the other end of the spectrum, bringing the latency down to microseconds only slightly improves performance. JITServer performs very well for latencies in the hundreds of microseconds which are typical in cloud datacenters. Performance in high-latency scenarios can be further improved by increasing the maximum number of client compilation threads (see Section 3.1). In our future work, we plan to investigate using remote JIT compilation in high-latency environments such as edge computing.

Summary of Results Our evaluation shows that:

- Remote JIT cannot be fully efficient without caching - it often increases total system-wide CPU cost (by up to 21%).
- Caching compiled code allows JITServer to reduce cluster-wide resource usage (by up to 77% for CPU and up to 62% for memory) and increase application density.
- JITServer significantly reduces application start-up and warm-up times (by up to 58% and 87% respectively) and memory footprint, especially in smaller containers.
- Caching dramatically improves JITServer scalability and allows it to effectively handle more concurrent clients and tolerate significantly higher network latency (up to 8 ms).

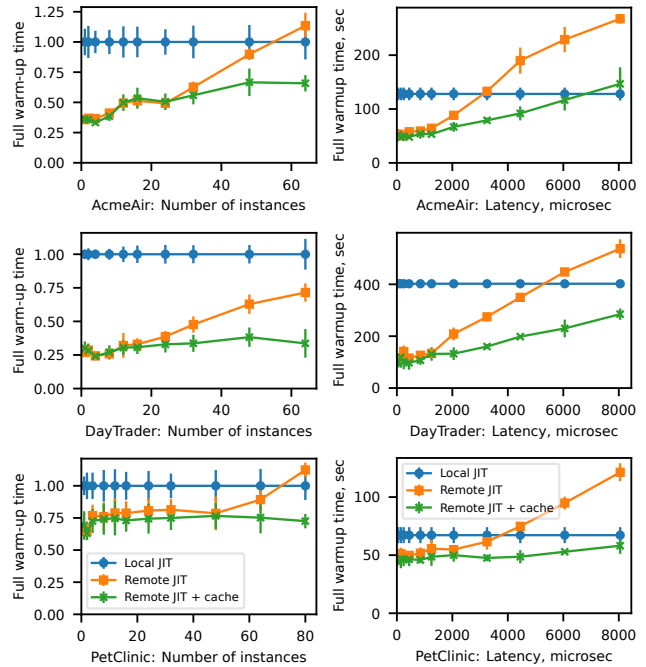


Figure 10: Scalability
(lower is better)

Figure 11: Effect of network latency
(lower is better)

5 Conclusion and Future Work

In this paper we explored JIT compiler disaggregation as a means to improve the performance and memory utilization of JVMs running in the cloud. We described JITServer, our disaggregated JIT compiler implementation in a production grade JVM, which supports the caching and reuse of compiled code across JVMs running on different hosts, and effectively amortizes JIT compilation costs over many client JVMs.

The experimental results showed excellent improvements in start-up time, warm-up time and memory footprint: JITServer is able to speed-up JVM start-up by as much as 58% and to reduce warm-up times by up to 87%, without a degradation in peak throughput. Moreover, the overall system-wide peak memory footprint is reduced by up to 62%, which should make it possible to reduce operational costs by increasing application density. Caching and reusing compiled code allows JITServer to reduce overall CPU cost by up to 77%, showing that JITServer enables cloud users to do more useful computation with less resources.

Our future work will explore prefetching of server-cached code and predicting hot methods to hide compilation latency and further reduce cold start times. We will study the trade-off between the performance of relocatable code and the JITServer cache hit rate. We will investigate ways to improve the JIT compiler in the disaggregated setting, e.g. utilizing profiling data from many clients, efficiently sharing compiled code across applications, and automatically sizing and scaling compilation resources. We plan to apply JITServer to other workloads such as FaaS, microservices, and data analytics.

A Artifact Appendix

Abstract

We provide the open source implementation of JITServer (as part of the OpenJ9 project), as well as the automated benchmarking platform that we used in our experimental evaluation. Our results have been successfully reproduced in artifact evaluation.

Scope

Our artifact can be used to run all the experiments described in Section 4 in order to validate the main claims in our paper, most importantly:

- JITServer can reduce application start time and warm-up time and system-wide CPU and memory usage for JVM-based applications running in containers with limited resources (which are common in the cloud).
- Caching dynamically compiled code at the JITServer is necessary to fully achieve the reduction in overall CPU usage and application start time.

Note that the experimental results are expected to be slightly different from the ones reported in this paper (even on the same hardware) since the OpenJ9 implementation has evolved since we conducted those experiments. However, the main conclusions should still hold.

Contents

The artifact consists of two parts:

1. The open source implementation of our system, which has been contributed to the OpenJ9 project. JITServer is implemented in ~25 KLOC of C++. The code is integrated into the rest of the OpenJ9 code base.
2. A set of scripts (Python, shell scripts, and Docker files) that automate the benchmark runs used in our evaluation and generate the resulting graphs. We also provide the logs generated by running the full set of experiments reported in this paper.

Hosting

All the parts of our artifact are open source and are hosted on GitHub. The OpenJ9 code base (including JITServer) is split across three separate repositories:

- Eclipse OpenJ9: <https://github.com/eclipse-openj9/openj9>;
- Eclipse OMR: <https://github.com/eclipse/omr>;
- OpenJDK extensions for OpenJ9 (we used JDK 8 in our experiments; newer JDK versions are also available): <https://github.com/ibmruntimes/openj9-openjdk-jdk8>.

The source code of our automated benchmarking platform is available at <https://github.com/AlexeyKhrabrov/jitserver-benchmarks>

Stable forks or branches of these repositories based on the 0.32.0 OpenJ9 release (with minor changes required by our

benchmarking platform, e.g. collecting additional statistics) that were used for artifact evaluation are available as follows:

- OpenJ9: <https://github.com/AlexeyKhrabrov/openj9/tree/atc22ae> (commit 724db2932e5f0abb);
- OMR: <https://github.com/AlexeyKhrabrov/omr/tree/atc22ae> (commit ab24b66665961405);
- JDK: <https://github.com/AlexeyKhrabrov/openj9-openjdk-jdk8/tree/atc22ae> (commit 0b8b8af39a5f1f2f);
- Benchmarks: <https://github.com/AlexeyKhrabrov/jitserver-benchmarks/tree/atc22ae> (commit 8360d90b89744ad1)

Requirements

While JITServer supports a wide range of Linux platforms, our benchmarking setup assumes Ubuntu 18.04 as the OS. It should be possible (although not necessarily easy) to tweak it to work on other Linux distributions. Newer Ubuntu versions might require downgrading to an older GCC version (OpenJ9 currently officially supports GCC 7, but should also support GCC 10). Different Linux distributions will need more tweaks, namely different ways of installing prerequisite packages.

Running the largest experiments reported in this paper requires a cluster of 11 machines with 16 CPU cores each, connected with a 10 Gbit/s network (or at least 1 Gbit/s) with round-trip latency between machines in the low hundreds of microseconds or less. Alternatively, the experiments can be run in a public cloud such as AWS on a cluster of virtual instances with roughly equivalent resources. The benchmark setup requires `sudo` permissions on all the machines. The required amount of storage space is approximately 30 GB on each node (not including the OS).

The hardware and software environment that we used in the experimental setup in our evaluation is described briefly in Section 4 and in more detail in the README document: <https://github.com/AlexeyKhrabrov/jitserver-benchmarks/tree/atc22ae#environment-used-in-our-evaluation>.

Usage

Detailed instructions describing how to set up and run the benchmarks and generate the results can be found in the README of the artifact repository: <https://github.com/AlexeyKhrabrov/jitserver-benchmarks>

Acknowledgements

We thank the anonymous reviewers and the shepherd for their feedback that helped us improve the paper, as well as the anonymous AEC members for their help in identifying and resolving issues in our artifact submission. We are grateful to the OpenJ9 developer community for their help in contributing

our code to the OpenJ9 project. This research was supported in part by IBM CAS Canada and an NSERC CRD grant.

References

- [1] AcmeAir sample and benchmark. <https://github.com/blueperf/acmeair-monolithic-java>.
- [2] AdoptOpenJDK OpenJ9 official image - Docker Hub. https://hub.docker.com/_/adoptopenjdk.
- [3] Amazon EC2 T2 instances. <https://aws.amazon.com/ec2/instance-types/t2/>.
- [4] Apache OpenWhisk runtimes for Java. <https://github.com/apache/openwhisk-runtime-java>.
- [5] Azul cloud native compiler. <https://www.azul.com/products/intelligence-cloud/cloud-native-compiler/>.
- [6] Eclipse OpenJ9. <https://www.eclipse.org/openj9/>.
- [7] GraalVM native image. <https://www.graalvm.org/reference-manual/native-image/>.
- [8] Java EE7: DayTrader7 sample. <https://github.com/wasdev/sample.daytrader7>.
- [9] JEP 295: Ahead-of-time compilation. <https://openjdk.java.net/jeps/295>.
- [10] JEP 410: Remove the experimental AOT and JIT compiler. <https://openjdk.java.net/jeps/410>.
- [11] Open Liberty official image - Docker Hub. https://hub.docker.com/_/open-liberty.
- [12] SPECjbb2015 benchmark. <https://www.spec.org/jbb2015/>.
- [13] SPECjEnterprise2018 Web Profile benchmark. <https://www.spec.org/jEnterprise2018web/>.
- [14] SPECjvm2008 benchmark. <https://www.spec.org/jvm2008/>.
- [15] Spring PetClinic sample application. <https://github.com/spring-projects/spring-petclinic>.
- [16] D. Bhattacharya, K. B. Kent, E. Aubanel, D. Heidinga, P. Shipton, and A. Micic. Improving the performance of JVM startup using the shared class cache. In *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, PACRIM, pages 1–6, 2017.
- [17] Guangyu Chen, Byung-Tae Kang, Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Rajarathnam Chandramouli. Studying energy trade offs in offloading computation/compilation in Java-enabled mobile devices. *IEEE Trans. Parallel Distrib. Syst.*, 15(9):795–809, September 2004.
- [18] Guilin Chen, Byung-Tae Kang, Mahmut T. Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Rajarathnam Chandramouli. Energy-aware compilation and execution in Java-enabled mobile devices. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*, page 34. IEEE Computer Society, 2003.
- [19] Ben Corrie and Hang Shao. Class sharing in Eclipse OpenJ9. <https://developer.ibm.com/tutorials/j-class-sharing-openj9/>, 2018.
- [20] Grzegorz Czajkowski, Laurent Daynès, and Nathaniel Nystrom. Code sharing among virtual machines. In *Proceedings of the 16th European Conference on Object-Oriented Programming, ECOOP '02*, page 155–177, Berlin, Heidelberg, 2002. Springer-Verlag.
- [21] Bertrand Delsart, Vania Joloboff, and Eric Paire. JCOD: A lightweight modular compilation technology for embedded Java. In *Proceedings of the Second International Conference on Embedded Software, EMSOFT '02*, page 197–212, Berlin, Heidelberg, 2002. Springer-Verlag.
- [22] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyst: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20*, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.
- [23] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20*, page 45–59, New York, NY, USA, 2020. Association for Computing Machinery.
- [24] Kiyokuni Kawachiya, Kazunori Ogata, Daniel Silva, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Cloneable JVM: A new approach to start isolated Java applications faster. In *Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE '07*, pages 1–11, New York, NY, USA, 2007. ACM.

- [25] Joel Koshy, Ingwar Wirjawan, Raju Pandey, and Yann Ramin. Balancing computation and communication costs: The case for hybrid execution in sensor networks. *Ad Hoc Networks*, 6(8):1185–1200, 2008.
- [26] Han B. Lee, Amer Diwan, and J. Eliot B. Moss. Design, implementation, and evaluation of a compilation server. *ACM Trans. Program. Lang. Syst.*, 29(4):18–es, August 2007.
- [27] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 383–400, Berkeley, CA, USA, 2016. USENIX Association.
- [28] Matt Newsome and Des Watson. Proxy compilation of dynamically loaded Java classes with MoJo. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, LCTES/SCOPES '02, page 204–212, New York, NY, USA, 2002. Association for Computing Machinery.
- [29] Radu Teodorescu and Raju Pandey. Using JIT compilation and configurable runtime systems for efficient deployment of Java programs on ubiquitous devices. In *Proceedings of the 3rd International Conference on Ubiquitous Computing*, UbiComp '01, page 76–95, Berlin, Heidelberg, 2001. Springer-Verlag.
- [30] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Re-playable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.
- [31] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: Application initialization at build time. *Proc. ACM Program. Lang.*, 3(OOPSLA):184:1–184:29, October 2019.
- [32] Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. ShareJIT: JIT code cache sharing across processes and its practical implementation. *Proc. ACM Program. Lang.*, 2(OOPSLA):124:1–124:23, October 2018.



RIKER: Always-Correct and Fast Incremental Builds from Simple Specifications

Charlie Curtsinger
Grinnell College

Daniel W. Barowy
Williams College

Abstract

Build systems are responsible for building software correctly and quickly. Unfortunately, traditional build tools like `make` are correct and fast only when developers precisely enumerate dependencies for every incremental build step. Forward build systems improve correctness over traditional build tools by discovering dependencies automatically, but existing forward build tools have two fundamental flaws. First, they are incorrect; existing forward build tools miss dependencies because their models of system state are incomplete. Second, they rely on users to manually specify incremental build steps, increasing the programmer burden for fast builds.

This paper introduces RIKER, a forward build system that guarantees fast, correct builds. RIKER builds are easy to specify; in many cases a single command such as `gcc *.c` suffices. From these simple specifications, RIKER automatically discovers fast incremental rebuild opportunities. RIKER models the entire POSIX filesystem—not just files, but directories, pipes, and so on. This model guarantees that every dependency is checked on every build so every output is correct.

We use RIKER to build 14 open source packages including LLVM and memcached. RIKER incurs a median overhead of 8.8% on the initial full build. On average, RIKER’s incremental builds realize 94% of `make`’s incremental speedup with *no manual effort* and *no risk of errors*.

1 Introduction

Build systems specify how code and other assets should be transformed into executable software. They capture compilation procedures left unstated in the source code itself. Build systems make the process of building software more reliable, since they free programmers from having to reproduce long sequences of build commands after making a change.

To be useful, build systems should satisfy two goals. First, builds must be correct: running a build should always have the same effect, whether code was built previously or not. Second, builds must be fast: they should perform the minimum amount of work required to update a build in response to a

change. These goals are often in tension. Builds that are simple to specify expose few opportunities for fast incremental updates, while complex incremental builds are more likely to be incorrect. To illustrate this challenge, we begin with an example build specification for `make`, one of the earliest and most widely-used build tools [8].

```
program ①: main.c x.c x.h y.c y.h ②
gcc -o program main.c x.c y.c ③
```

A `make` build specification, written in a Makefile, lists a collection of *build rules*. The example above shows a rule that produces a single program from three source files and two include files. Each target is composed of three parts: ① a *target name*, usually the name of an output file; ② a list of *dependencies*, which `make` calls “prerequisites,” required to produce the target; and ③ a *recipe* that includes the build commands needed to produce the target from the dependencies.

At the start of each build, `make` compares the last modification time of every target with its dependencies. When a dependency is newer than the target, `make` runs the recipe to update the target. When a dependency does not exist, `make` recursively runs the rule that builds the dependency.

The key insight in `make`’s design is that developers rarely change an entire codebase between builds. Rebuilds can run faster by doing work proportional to the number of changed dependencies, not the total number of files. In the simplest case, when no dependencies change, `make` does nothing at all.

It is easy to see that the example Makefile is correct because all the dependencies are present and only a single build command is needed. Unfortunately, this build is also inefficient. Changing `x.c` will cause `gcc` to recompile all three `.c` files, even though `y.c` and `main.c` are unchanged. The cause of this inefficiency is that the build is monolithic: there is only one target that depends on all source files, so `make` must run a full build when any source file changes.

Monolithic builds are prohibitively expensive for larger projects. For example, a full build of LLVM takes nearly 20 minutes when run in parallel on a typical developer workstation—far too long for a developer to wait to test a

small code change. Not surprisingly, large projects often have incremental build specifications. To produce an incremental `make` build, developers must break a specification into smaller rules, exposing intermediate targets. The following modifies the original `Makefile` so that it can be built incrementally.

```
program: main.o x.o y.o
    gcc -o program main.o x.o y.o
main.o: main.c x.h y.h
    gcc -c -o main.o main.c
x.o: x.c
    gcc -c -o x.o x.c
y.o: y.c y.h
    gcc -c -o y.o y.c
```

The updated specification states how to build a `.o` file from each source file, and how those `.o` files are combined into the final target, `program`. The new `Makefile` describes the same work that `gcc` performs internally; internal steps have simply been exposed so that `make` can run them or skip them during an incremental rebuild. With this `Makefile`, modifying `x.c` no longer rebuilds every output. Instead, the build generates a new `x.o`, linking it with the other `.o` files already on disk.

This `Makefile` also illustrates the dangers inherent in more complex build specifications: missing dependencies. Suppose `x.c` includes `x.h`. The original `Makefile` is correct, but the refactored one is not because changing `x.h` does not trigger a rebuild of `x.o` as it should. The implication of such a bug is that a developer with a previously-built working copy could end up with different output than another developer who starts with a clean copy of exactly the same source code.

An incremental build must be *consistent* with the full build; it should always produce output that could have come from a full build. `Make` and build tools like it do not guarantee this property because they do not check for missing dependencies. These errors occur with alarming frequency. A recent study showed that more than two-thirds of the open-source programs analyzed had serious build specification errors [28].

To address build errors, recent work proposes the idea of a *forward build system* [29]. Build systems like `make` are “backward” because evaluation proceeds from the final output rules, recursively building dependencies as needed. A forward build specification instead lists a sequence of commands, in order, that perform a full build. Critically, forward build systems discover dependencies automatically using program tracing instead of asking users to enumerate dependencies. On rebuild, a forward build system runs only the commands necessary to update the build, just as `make` does. When correctly designed and implemented, forward build systems guarantee correctness because they never miss dependencies.

Unfortunately, prior forward build systems miss dependencies because they fail to account for the complexity of real builds. Worse, they require users to manually specify incremental build steps. As a result, prior forward build systems are neither automatically correct nor automatically fast.

This paper introduces RIKER, a forward build system that

delivers the benefits of an incremental build system with the simplicity of monolithic specifications. RIKER substantially advances the state of the art in forward build systems by using a completely different algorithmic approach. With RIKER, efficient incremental builds can be specified using a single build command like `gcc *.c`.

RIKER captures dependencies on directories, pipes, links, and sockets—not just files—ensuring that builds are correct. RIKER infers fine-grained steps from monolithic build specifications, ensuring that builds are fast. In the above example, RIKER captures the execution of the C compiler (`cc1`), assembler (`as`), and linker (`ld`), can run these commands incrementally, and in many cases in parallel, to update the build. In short, RIKER makes it possible for users to specify builds that are simple, correct, and efficient.

Contributions

Tracing and TraceIR. We present a high-performance system call tracing mechanism that generates dependence-checking programs in the novel TraceIR language (§4). TraceIR captures all of a build’s state interactions—including, but not limited to, paths, files, directories, and pipes. TraceIR facilitates correct handling of circular and temporal dependencies, complexities that occur in real builds.

RIKER Build Algorithm. The RIKER algorithm performs efficient incremental builds by mixing emulation of previously-recorded TraceIR with re-execution of commands whose dependencies have changed (§5).

Implementation and Evaluation. We present an implementation of RIKER for Linux, and evaluate this implementation by using it to build 14 real-world software projects (§6). Our evaluation shows that RIKER is a significant advance over the previous state of the art in forward build tools, automatically performing incremental builds that are competitive with manually-written `make` builds. RIKER is available under an open-source license at <https://rkr.sh>.

2 Related Work

Build systems have evolved significantly since `make`’s introduction in 1976 [8, 21]. However, all build systems share the two goals we identify in the introduction: builds must be correct and they must be fast. Build systems differ substantially in their configuration, the precision of their dependency tracking, how changes are detected, and the level of manual effort required to use them. These differences, which we discuss below, are summarized in Table 1.

Backward Build Systems. The `make` build system and most of its successors are *backward build systems*. With a backward build system, the user writes a rule for each target produced by the build, lists the dependencies required to produce the target, and provides the commands required to create the target from its dependencies. When a target’s dependency does

		Source & Build Language		Dependencies			Incremental Builds	
		Build System	Language-Agnostic	No DSL	Precise	General	Automatic	Dynamic
Backward	Make, Ninja, Shake, Tup	✓						
	Vesta	✓		✓		✓*	✓	
	CMake, Ant/Maven, SCons					✓ ^λ		✓ ^λ
	Pluto					✓ ^λ	✓	✓ ^λ
	Bazel, Buck			✓	✓*		✓*	✓ ^λ
Forward	Memoize, Fabricate	✓	✓*	✓		✓		
	Rattle	✓		✓		✓	✓	
	RIKER	✓	✓	✓	✓	✓	✓	✓

Table 1: A comparison between RIKER and prior build tools. A build system is *language agnostic* if it can build projects written in any source language or combination of languages, and has *no DSL* if builds can be specified in a general-purpose language that is executable independent of the build system. A build system is *precise* if it ensures that a specification captures all dependencies. It is *general* if it allows cyclic dependencies, anti-dependencies, and dependencies on non-file objects. A build system is *automatic* if it discovers dependencies or incremental builds without manual specification. A tool that supports *dynamic* incremental builds can discover dependencies while a build runs. A ✓* indicates partial support, and ✓^λ indicates that a feature’s support is language-specific.

not exist, a backward build system calls the rule that produces the dependency. Such build specifications are essentially an edge-by-edge encoding of a dependence graph.

Make, Tup [27], Shake [22], and Ninja [1] are all examples of backward build systems that require a dependence graph encoding. Writing build specifications for these tools can be burdensome. CMake [4], Ant/Maven [2, 3], SCons [26], and Pluto [5] reduce this burden by providing standard templates. Templates encode common build procedures like producing an executable from a collection of C source files. These tools automatically discover dependencies and run incremental builds, but only for supported languages. Users must provide extensions to these build tools before they can use them to build projects that use unsupported languages.

The early and innovative Vesta build system specifies builds in a general-purpose modeling language [12, 13]. Although users encode dependencies manually, Vesta uses a form of black-box tracing at the filesystem layer to identify and cache unspecified build outputs for incremental speedups. As Vesta is also a source control management tool, it can correctly reuse cached build objects in a distributed setting. Vesta can only skip work for explicitly enumerated build steps.

Buck [7] and Bazel [11] focus on ensuring that dependencies are *precise* by intentionally failing when any dependency is not explicitly provided. This is in direct contrast to Make, Tup, and Shake, where missing dependencies can lead to incorrect builds. Like CMake, Buck and Bazel simplify the process of specifying incremental builds with templates for supported languages. Buck, Bazel, and Pluto also offer some degree of dynamism when building; they can discover some additional dependencies or prune work as the build proceeds.

RIKER differs from all these tools because it precisely captures fine-grained dependencies and provides automatic speedups without manual specification, and it does so for projects written in any programming language.

Forward Build Systems. Memoize [19], Fabricate [14], and Rattle [29] use tracing to discover dependencies from a sequence of build commands that should run in order. These systems guarantee precise dependencies while remaining language agnostic. However, all of them are limited to modeling file state. A change to unsupported state, like a directory, does not trigger a rebuild, producing an incorrect rebuild. As we discuss in this work, correct builds must model not just files, but inode metadata, directories, symbolic links, hard links, pipes, and sockets. Correct build systems must also model the *absence* of such state. Existing forward build tools also have limited ability to produce fast incremental builds. Compiler drivers like `gcc` launch many separate sub-commands to compile, assemble, and link programs. Prior forward build systems require users to enumerate incremental build steps, which makes writing build specifications more difficult.

Like the above tools, RIKER is a forward build system. RIKER substantially improves the state of the art by automatically inferring fine-grained build steps. RIKER can directly invoke sub-commands called by wrapper programs like `gcc`. Sub-command execution is possible because RIKER’s dependence tracking is both precise and complete.

Separation of Concerns. Many build tools tackle additional tasks, such as platform detection or compilation in a distributed setting. For example, Autoconf [9] and Automake [10] detect characteristics of the local system to generate build configurations, usually as a precursor to running `make`. Buck, Bazel, Vesta, and CloudBuild [6] offer support for distributing builds on cloud services. We regard these objectives to be orthogonal to this work. Since a RIKER build specification is just a program, it can include configuration steps. And we suspect that RIKER’s precise dependency tracking may make it easier to distribute builds across machines, although we have not explored this topic.

RIKER’s design is strongly influenced by the UNIX philosophy to make each program “do one thing well” [20]. There-

fore, we focus this work narrowly on one problem: to build software correctly and quickly. This paper provides evidence that the key to building software correctly and quickly is the accurate detection and handling of changed dependencies, regardless of language. The fact that RIKER can be used to orchestrate compilation *for* any language, using a specification written *in* any language, is a useful property that emerges from the exclusive pursuit of this sole concern.

3 Overview

RIKER builds software using a simple specification called a Rikerfile. A Rikerfile is typically a short shell script, although it can be any executable that performs a full build. RIKER is designed to spare developers the error-prone work of specifying dependencies. Instead, RIKER discovers them automatically as a Rikerfile executes. RIKER uses system call tracing to capture all of a build’s stateful interactions, which it records in a novel intermediate representation called TraceIR. A TraceIR *transcript* is a *program* that describes a build’s dependencies, operations, and effects (see §4).

When a user requests a rebuild, RIKER evaluates the stored transcript instead of running a full build (see §5). Evaluation updates an in-memory model of system state; the model captures the effects that *would occur* if a full build were run again. Every statement in the TraceIR language is a predicate: whenever the modeled result of a TraceIR statement differs from the observed outcome of the previous build, a dependency has changed. Any command whose dependencies change must be re-executed to update the build.

During TraceIR evaluation, RIKER must decide: should a command be *executed*, or can it be *skipped*? If a command is executed, RIKER actually runs the command using the `execve` system call, tracing its execution and replacing its old TraceIR statements in the transcript. If a command is skipped, RIKER instead *emulates* the command, replaying its effects (if it has any) in the model and only *syncing* those effects to the filesystem at the end of the build or when an executed command needs to observe them. In general, emulation is orders of magnitude faster than execution. RIKER runs fast incremental builds by skipping commands whenever it can; the number of executed commands is roughly proportional to the number of changes.

RIKER repeatedly re-evaluates the entire trace until no changes are found. Re-evaluation is necessary because an executed command can change a dependency for another command. Instead of conservatively running all commands that *might* observe changes, RIKER defers the decision to execute until it can prove that a command *must* execute. Once no commands must execute, the build is “up to date” and RIKER saves the updated transcript for use in the next rebuild.

3.1 Examples

In this section, we highlight three scenarios from the working example (from §1) that illustrate RIKER’s operation. The

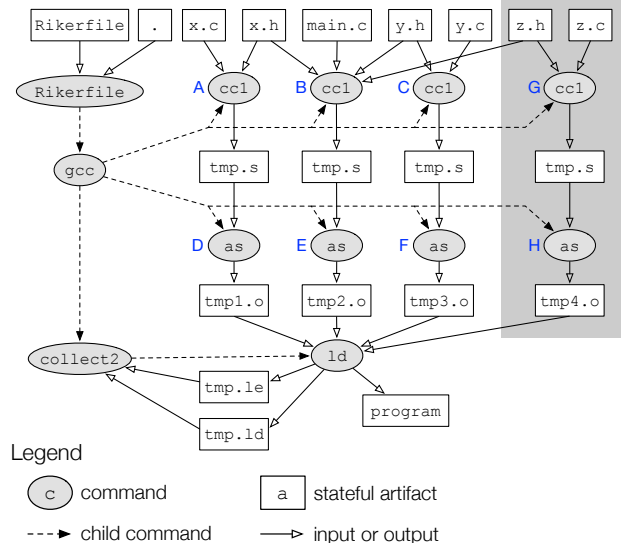


Figure 1: A dependency graph for the running example. Dashed edges show commands launching child commands, and solid edges indicate inputs and outputs. The grey box contains the modification induced by adding `z.h` and `z.c` to the build in **Scenario 2**.

following Rikerfile specifies the example build:

```
#!/bin/sh
gcc -o program *.c
```

Scenario 1: Running the first build. A user runs `rkr` to perform a build. When `rkr` is invoked with no saved state RIKER starts a full build by executing Rikerfile.

Figure 1 is a dependence graph for the running example. Oval vertices represent *commands*, which correspond to programs run via `exec` system calls. Rectangular vertices represent stateful *artifacts* such as files or directories. Dashed edges indicate that a parent command *launched* a child command. Solid edges indicate a command’s input or output.

In contrast to many other build systems, RIKER’s build algorithm does not store build information as a dependence graph. Dependence graphs lack critical temporal information needed to disambiguate rebuild logic that arises from circular dependencies. We show Figure 1 to illustrate that even simple builds have complex dependence structure that RIKER can exploit. In fact, for clarity, Figure 1 omits a great deal: dependencies on system includes, shared libraries, the executable files for each command, and intermediate states of artifacts written multiple times during the build.

When the Rikerfile command launches `gcc`, it launches three instances of `cc1` in turn. Each `cc1` instance compiles a `.c` file (and any included `.h` files) to a `.s` assembly file. `gcc` also launches three instances of `as` to produce `.o` object files from each `.s` input. Finally, `gcc` launches the `collect2` command, which launches the linker `ld`. `ld` redirects `stdout` and `stderr` to temporary files which can cause `collect2` to rerun the linker with different options. Note the cycle between `collect2` and `ld`; this dependence cycle is present in every build that uses `gcc`. Also observe that `gcc` repeatedly reuses

the same `tmp.s` temporary file, truncating it at the start of each `cc1` execution. File reuse and cyclic dependencies occur frequently in builds.

Instead of a dependence graph, RIKER operates over a TraceIR transcript. RIKER translates every intercepted system call into a sequence of TraceIR statements. The following transcript excerpt is generated during our example build:

```
1 sh_0 = Launch(rkr, "sh Rikerfile", [...])
2 r16 = PathRef(sh_0, CWD, ".", r--)
3 ExpectResult(sh_0, r16, SUCCESS)
4 MatchMetadata(sh_0, r16,
  [uid=100, gid=100, type=dir, perms=rwxrwxr-x])
5 MatchContent(sh_0, r16, [dir: {"Rikerfile",
  "main.c", "x.c", "x.h", "y.c", "y.h"}])
```

The transcript above records that RIKER launched the `Rikerfile` program (line 1). Linux resolved the current working directory path (line 2) without error (line 3), and the resolved directory has the given metadata and directory entries. During tracing, RIKER interprets all transcribed information as what *should happen* during the normal course of a build. While `Rikerfile`'s access of the current working directory's path resolved without error in this example, path resolution failing with `ENOENT` is a normal occurrence whose outcome must be recorded. For example, when the user types `gcc` at the prompt, UNIX may first try to access `gcc` in `~/bin`, which will fail if `~/bin` does not contain `gcc`. This behavior is the result of the fact that UNIX searches the user's `$PATH` during resolution [16]. The observed sequence of failures is a build dependency, and any change in failures implies that a build must rerun. We call failing resolutions *anti-dependencies*.

In the next section, we examine how the above transcript guides a rebuild after a user makes a code change. We defer discussion of TraceIR semantics to Section 4.3.

Scenario 2: Adding a file. Suppose a user adds files `z.c` and `z.h` and modifies `main.c` to include `z.h`. The user then runs `rkr` to update the build. The grey box in Figure 1 shows the effect of the change on the build's dependence graph. An efficient build system should not rebuild files unrelated to a change. Here, `tmp1.o` and `tmp3.o` do not need updating since they do not depend—even transitively—on any of the changes. At the very least, `cc1` and `as` should be called to compile `main.c` and `z.c`, and `collect2` and `ld` should be called to link the output to our preexisting object files. In fact, the very least is exactly what RIKER does here.

RIKER performs an incremental rebuild of the example by evaluating the TraceIR from the previous build. We assume the user does not change ownership or permissions for the current directory, so lines 1–4 evaluate just as before. However, line 5, which depends on directory contents, reports a change because the current directory contains the new files `z.c` and `z.h`. RIKER therefore reruns and traces the `Rikerfile`. When the `Rikerfile` command is rerun, the `Rikerfile`'s portion of the transcript is replaced with newly generated TraceIR.

Although rerunning the `Rikerfile` might seem to imply

that the entire build will run again, this is not the case. When `Rikerfile` launches `gcc`, RIKER lets the execution proceed (also under tracing) because `gcc`'s arguments, which now include `z.c`, also change. However, RIKER *skips* execution of the commands labeled A, C, D, and F in Figure 1, emulating them from the trace instead.

Command skipping. Let us examine the first command that RIKER skips, the instance of `cc1` labeled A in Figure 1:

```
1 cc1_1 = Launch(gcc_0, "cc1 x -o tmp.s", [...])
2 r71 = PathRef(cc1_1, CWD, "x.c", r--)
3 ExpectResult(cc1_1, r71, SUCCESS)
4 MatchMetadata(cc1_1, r71,
  [uid=100, gid=100, type=file, perms=rw-rw-r--])
5 MatchContent(cc1_1, r71,
  [mtime=1619457130, hash=3c6ea, cached=false])
6 r75 = PathRef(cc1_1, r3, "tmp.s",
  -w- truncate create (rw-rw-rw-))
7 ExpectResult(cc1_1, r75, SUCCESS)
8 UpdateContent(cc1_1, r75, [hash=054521])
```

The key observation is that emulation of this command's transcript—which RIKER always does before concluding that a command must run—detects no changes. `cc1` reads `x.c` (lines 4–5) and writes to a temporary file, `tmp.s` (lines 6–8). The file `x.c` is unchanged. Although `tmp.s` was created by `gcc` and is reused by every `cc1` process started by `gcc`, there is no dependency on that file's content because `cc1` truncates the file without reading it (line 6). Because RIKER detects no changes, it can emulate rather than execute `cc1`. Similar reasoning allows RIKER to skip C, D, and F.

Scenario 3: Making an inconsequential change. Suppose a comment is added to `x.c` and the build is run again. This change has no effect on the final compiled program and an ideal build system should do almost nothing. Referring again to the previous trace, RIKER detects a change (line 5) for `cc1` because `x.c` changes. However, RIKER correctly halts the build without ever running `as`, `collect2`, or `ld`.

Here is an excerpt of the `as` command's transcript:

```
1 as_1 = Launch(gcc_0, [as -o tmp1.o], [...])
2 r26 = PathRef(as_1, r3, "tmp1.o",
  rw- truncate create (rw-rw-rw-))
3 ExpectResult(as_1, r26, SUCCESS)
4 r27 = PathRef(as_1, r3, "tmp.s", r--)
5 ExpectResult(as_1, r27, SUCCESS)
6 MatchMetadata(as_1, r27, [uid=100,
  gid=100, type=file, perms=rw-----])
7 MatchContent(as_1, r27, [mtime=1619458806,
  hash=10732f, cached=true])
8 UpdateContent(as_1, r26, [mtime=1619458806,
  hash=3814e7, cached=true])
```

The `as` command reads `tmp.s` and writes to `tmp1.o`. RIKER concludes that `cc1`'s output `tmp.s` is unchanged (lines 6–7), even though the `mtime` is different. In RIKER, `mtimes` provide a fast path for change detection: a file's content is unchanged if its `mtime` is unchanged, or if its content produces the expected hash. Finally, we see `as` writes to `tmp.o` (line 8). Because `as` is unchanged, RIKER can instead restore the `tmp.o` file from its cache and skip `as`.

3.2 Summary

RIKER runs efficient incremental builds from simple specifications, even those with a single command. Throughout this section we never once needed to change the `Rikerfile`, even though the build's dependencies changed with the addition of new files; RIKER always ensures that incremental rebuilds produce the same effect as full builds. Finally, RIKER is language agnostic: it can be used to build programs written in any language using any executable build specification.

4 Tracing and TraceIR

This section defines terms that we use to describe RIKER's operation, describes RIKER's system call tracing mechanism, and introduces the novel TraceIR language that RIKER uses to record dependency information.

4.1 Definitions

An *artifact* represents system state such as a file, a directory, or a pipe. A *version* represents an artifact's state at a point in time during the build. Every artifact has both content and metadata versions. Two content versions are *identical* if and only if they contain the same bits. Two metadata versions are identical if and only if their permission bits and ownership are the same.

A *command* is a program that accesses or modifies artifacts, and may launch additional commands. A *dependency* is any version made visible to a command through a system call. A command is *changed* if its dependencies are not identical to the versions observed during the previous run, or if the outcome of an operation like path resolution differs from what was observed during the previous build (e.g. a file referenced during the last build no longer exists).

Versions are *equivalent* if they are identical, or if they could be written to the same artifact by two executions of a non-deterministic command given equivalent inputs. Outputs from different executions of a command given equivalent inputs are interchangeable, even if those outputs are not identical.

A *build specification* is any program that encodes a sequence of commands to run. A *build system* is a program that evaluates a build specification to perform a *build*. A *full build* executes all of the commands in the specification, while an *incremental build* produces the same effect while executing only a subset of commands.

When executing a command, RIKER records the dependencies and effects of each command in the TraceIR language. RIKER evaluates recorded TraceIR on subsequent builds to determine whether the command has changed. A command can be *skipped* if it is unchanged, otherwise it must be *executed*.

4.2 Tracing Implementation

RIKER executes commands under system call tracing to gather a complete set of dependencies. RIKER uses `ptrace` [17] to intercept system calls. Only calls related

to filesystem interaction, file descriptor management, and process creation—75 system calls in total—matter for dependency tracking. RIKER combines `ptrace` with `seccomp-BPF` [25] filters to avoid tracing irrelevant system calls. Nevertheless, tracing has high overhead even with filtering. To reduce overhead, RIKER injects a shared library into each build command to intercept calls to frequently-used `libc` system call wrappers like `open` and `stat` without incurring `ptrace` overhead. This approach is inspired by RR [24], with additional support for passing system call arguments through shared memory. For a build of `memcached`—a typical C project built with `gcc`—RIKER's injected library handles 95% of system calls (`ptrace` handles the remaining 5%). For `memcached`, shared library tracing reduces RIKER's full-build overhead from 1.81s (16.9%) to just 0.84s (7.8%) with no loss of tracing precision.

The next section describes the TraceIR language. Section 4.4 shows how system calls are translated to TraceIR.

4.3 The TraceIR Language

TraceIR is an executable, linear representation of a build's behavior. RIKER generates TraceIR from system call traces, and detects changes by evaluating TraceIR against a model of the filesystem. The TraceIR language describes the dependencies that are visible to each command during the build, as well as the side effects of each command's execution.

Design considerations. The design of TraceIR is guided by three important requirements. First, tracing must be *complete*, capturing dependencies on all artifacts. A missed dependency could lead to an incorrect build. Second, TraceIR records events *serially*, even if those events come from processes that run in parallel during the build. Recording events serially imposes a temporal ordering [15] on filesystem interactions. A temporal ordering makes the relationship between an access of an artifact and its corresponding version unambiguous at any given point in a build, even when that artifact is read and written concurrently. Finally, a TraceIR program represents *a single observed path of execution* for a command. If a command's dependency changes, RIKER will execute and trace the command to discover its new behavior.

Language elements. Table 2 shows the datatypes of the TraceIR language, and Table 3 shows nearly all of the statements in the TraceIR language. We omit three low-level operations for clarity. Return types are `BOOL` where omitted. A TraceIR program is a sequence of statements, each associated with a command *c*. Statements fall into four logical groups:

Artifact accesses establish a reference to an artifact. A reference can resolve successfully or result in an error. Some accesses are side effecting (e.g. to capture the behavior of `open` with the `O_CREAT` flag).

State checks record the state observed by a command *c* the last time it executed. If a state check fails, *c* observes a change and must re-execute.

TraceIR Data Types	
BOOL, INT, STRING	Normal primitive types
[T]	A list of elements of type T
OUTCOME	Success or a POSIX error code
REF	Reference to an artifact or OUTCOME
CMD	A build command
FLAGS	Flags associated with a file access
METADATAVERSION	An artifact's metadata
CONTENTVERSION	An artifact's content
SPECIALREFID	A special artifact ID: ROOT, CWD, ...

Table 2: TraceIR data types.

State updates record the effect a command had on global state, such as writes to file content or metadata, or the creation and removal of directory entries.

Command updates record command creation, termination, and when a command waits for another to exit. These statements capture the semantics of `execve`, `exit`, `wait`, and related system calls.

4.4 Generating TraceIR Transcripts

RIKER translates each traced system call to a sequence of TraceIR statements called a *transcript*. The transcript below is generated when command *c* issues a successful `stat("input", &statbuf)` call.

```

1 r1 = SpecialRef(c, CWD)
2 r2 = PathRef(c, r1, "input", ---)
3 ExpectResult(c, r2, SUCCESS)
4 MatchMetadata(c, r2, [uid=100, gid=100,
   type=file, perms=rw-r--r--])

```

Line 1 is a reference to the command's current working directory. Line 2 uses the reference to resolve the path to the file `input`. This `stat` call did not include any special flags so the fourth parameter is `---`. If the file were opened rather than `stated`, this field would request read and/or write permission. For `lstat`, a `nofollow` flag would signal that path resolution should not follow symbolic links.

Because `stat` succeeds, line 3 records a successful outcome. If `input` is removed before a future rebuild, the observed result will be `ENOENT` and RIKER will detect a change. Finally, because *c* observes `input`'s metadata, line 4 records the observed metadata. If the file's metadata is modified, a rebuild will detect a change.

Importantly, TraceIR also records anti-dependencies. Supposing the `stat` call originally failed with `ENOENT`, the generated TraceIR would have had the same first two lines, but line 3 would expect `ENOENT` and line 4 would be omitted. If `input` is present on rebuild, RIKER would observe that the result is not `ENOENT` and would detect a change.

5 Build Algorithm

The RIKER algorithm, shown in Figure 2, performs two tasks: it detects changes and updates the build by running commands affected by those changes. RIKER's build algorithm

TraceIR Statements	
Artifact Access	<code>PATHREF (c : CMD, b : REF, p : STRING, f : FLAGS) : REF</code> Command <i>c</i> resolves path <i>p</i> relative to <i>b</i> with flags <i>f</i>
	<code>SPECIALREF (c : CMD, id : SPECIALREFID) : REF</code> Command <i>c</i> references a special artifact (e.g. <code>/</code> , <code>cwd</code>)
	<code>FILEREF (c : CMD) : REF</code> <code>DIRREF (c : CMD) : REF</code> <code>PIPEREF (c : CMD) : REF, REF</code> <code>SYMLINKREF (c : CMD) : REF</code> Command <i>c</i> references new anonymous file, directory, etc.
State Checks	<code>EXPECTRESULT (c : CMD, r : REF, e : OUTCOME)</code> Command <i>c</i> expects <i>r</i> to resolve with outcome <i>e</i>
	<code>MATCHMETADATA (c : CMD, r : REF, e : METADATAVERSION)</code> <code>MATCHCONTENT (c : CMD, r : REF, e : CONTENTVERSION)</code> Command <i>c</i> expects <i>r</i> to have metadata or content <i>e</i>
	<code>EXITRESULT (c : CMD, child : CMD, n : INT)</code> Command <i>c</i> expects <i>child</i> to have exit code <i>n</i>
State Updates	<code>UPDATEMETADATA (c : CMD, r : REF, v : METADATAVERSION)</code> <code>UPDATECONTENT (c : CMD, r : REF, v : CONTENTVERSION)</code> Command <i>c</i> updates <i>r</i> with metadata or content <i>v</i>
	<code>ADDDIRENTRY (c : CMD, d : REF, e : STRING, a : REF)</code> Command <i>c</i> links artifact <i>a</i> as entry <i>e</i> in directory <i>d</i>
	<code>REMOVEDIRENTRY (c : CMD, d : REF, e : STRING)</code> Command <i>c</i> removes entry <i>e</i> from directory <i>d</i>
Command Updates	<code>LAUNCH (c : CMD, cmd : [STRING], rs : [REF]) : CMD</code> Command <i>c</i> launches child <i>cmd</i> with inherited references <i>rs</i>
	<code>JOIN (c : CMD, child : CMD)</code> Command <i>c</i> waits for child <i>child</i> to exit
	<code>EXIT (c : CMD, n : INT)</code> Command <i>c</i> exits with status <i>n</i>

Table 3: TraceIR Statements.

is always guided by a saved transcript. RIKER's fixed-point build algorithm repeatedly evaluates the build transcript, running changed commands while checking for changes made visible to other commands. The build terminates when no new changes are found.

Changes. TraceIR statements are predicates that express expectations about state at specific points during the build. An expectation can be in regard to the outcome of path resolution, the exit code of a child command, or the content or metadata of artifacts. Predicates are checked against the the filesystem model *M*. This model, a set of artifacts, records the effects of any prior TraceIR statements. The model is lazily populated with actual filesystem state; predicates that refer to artifacts that have not been modified during the build are checked against actual filesystem state. When a command's TraceIR predicate fails—that is, the expected state does not match the model *M*—the command is changed and must execute to update the build.

Phases. RIKER's build algorithm runs in *phases*. Each phase is an evaluation of an entire TraceIR build transcript *T* in the context of the model *M*. Trace evaluation, carried out by `EVALTRACE` (line 6 of `DOBUILD`), is repeated until the set of commands that must run, *R*, is empty.

```

DOBUILD(trpath)
1  i = 1, M = { }, R = { }
2  T = LOADTRACE(trpath)
3  if |T|==0 then T = { LAUNCH (rkr, Rikerfile, ...) }
4  repeat
5      M = { }
6      (M, T, D, R) = EVALTRACE(M, T, R, false)
7      R = PLAN(D, R)
8      i = i + 1
9  until |R|==0
10 SYNCALL (M)
11 if i > 1
12     (–, T, –) = EVALTRACE(M, T, R, true)
13 WRIETRACE(T)

EVALTRACE(M, T, R, post)
1  T' = nil, D = { }
2  Rbuild = { }, Rpost = { }
3  for t in T
4      c = CMDOF(t)
5      if c ∉ R
6          (ts, M, D, δbuild, δpost) = EVALSTMT(t, M, R, D, post)
7          if δbuild then Rbuild = Rbuild ∪ {c}
8          if δpost then Rpost = Rpost ∪ {c}
9          T' = T' @ ts
10 return (M, T', D, Rbuild ∩ Rpost)

```

Figure 2: RIKER’s build algorithm. DOBUILD repeatedly evaluates a TraceIR build transcript *T* in the model *M* until it observes no new changes. EVALTRACE evaluates a single pass through *T*, returning an updated model and trace, command dependence graph *D*, and set of commands that must run, *R*. @ concatenates two lists. At the end of the repeat-until loop, the build is up-to-date.

Emulation vs. Execution. RIKER performs incremental builds by mixing execution and emulation of build commands. RIKER *emulates* a command by evaluating its TraceIR to update the in-memory model *M*, but does not run the command. RIKER *executes* a command by running it with `exec(3)`. RIKER traces the system calls of the executing command to generate new TraceIR, and evaluates this new TraceIR to keep the in-memory model *M* in sync with filesystem state. Whether a command is emulated or executed depends on whether it was added to *R* during a previous phase (see §5.1).

Invariant. RIKER enforces the invariant that any command whose dependencies have changed will be executed; failing to do so results in an incorrect build. RIKER emulates all other commands instead of executing them.

Executing an unchanged command preserves correctness, but doing so is costly. We avoid unnecessary command executions whenever possible because they take orders of magnitude longer than emulating a command to recreate its effects. For example, when a command’s output is missing the file can be restored from a cached copy (see §5.2) instead of executing the command to recreate the file. We discuss the correctness of this approach in Section 5.5.

To avoid over-approximating *R* (the set of commands that will run) RIKER evaluates the build transcript *T* repeatedly (line 4). Repeated evaluation is necessary because change detection is a dynamic problem [5]. For example, suppose *T* contains two commands, *A* and *B*, and that *B* reads one of *A*’s outputs. Suppose *A* must run; does *B* need to run? The answer is “maybe.” If *A* produces the same output that it produced previously, then *B* does not need to run. We saw this exact situation in the overview’s example when adding a comment to a source file. It is safe to conservatively run *B* any time *A* runs, but RIKER can potentially save work by deferring the decision to run *B* until after *A*’s effects are observed.

Statement evaluation. At the heart of the build algorithm is the evaluation of TraceIR statements. The purpose of evaluating a statement is twofold: to update state and to detect changes. When a command is emulated, the only state updated is *M*. When a command is executed, both *M* and the actual filesystem are updated. The semantics of change detection depend on the specific TraceIR statement (see §4.3).

EVALSTMT (line 6) evaluates a TraceIR statement. Evaluation returns one or more TraceIR statements, an updated model *M*, a command dependence map *D* (see §5.3), and two flags denoting whether the statement observed a change, either a build or post-build change (see **Post-build checks** below). The returned trace statements are used in the next build phase. When a command is emulated, its trace steps are simply echoed back. We describe command execution in §5.1. After a transcript is evaluated, RIKER calls PLAN (line 7), which may mark extra commands to run (see §5.3).

Filesystem and model. RIKER begins each phase by initializing the model *M* (line 5 in DOBUILD). Emulated updates are discarded after each phase because those updates are replayed in subsequent phases, whereas changes resulting from execution are written directly to the filesystem. SYNCALL (line 10 of DOBUILD) writes all changes in the model *M* to the filesystem at the end of the build. This operation ensures the outputs from commands that did not need to run are written to disk.

Post-build checks. Desirable state left behind from a previous build can look like a change to a command run in an early part of a build. RIKER finishes every build with a series of *post-build checks* to avoid doing unnecessary work in this scenario. To illustrate, recall our working example from section 3 which runs the command `gcc -o program *.c`. Before performing any compilation steps, gcc will stat the program file, which does not yet exist. The stat system call produces the following TraceIR:

```

1  r8 = PathRef(gcc_0, CWD, "program", ---)
2  ExpectResult(gcc_0, r8, ENOENT)

```

Later in the build, program is created by the linker. As a result, immediately running the build again after completing the first full build would detect a change on line 2, and gcc would run. However, running gcc is unnecessary because the observed change is a byproduct of the build itself. Post-build

checks enable RIKER to skip over these changes by encoding multiple justifications to skip: a command is unchanged if all of its predicates match what was observed during the build, or if the predicates match state found *immediately after* the conclusion of the build.

The post-build phase (line 12 of DOBUILD) emulates all commands in T , but adds additional predicates to check post-build state. After running the post-build phase in the example build, the above TraceIR excerpt is extended to capture either outcome of the `stat` call:

```
1 r8 = PathRef(gcc_0, CWD, "program", ---)
2 ExpectResult(gcc_0, r8, ENOENT) [build]
3 ExpectResult(gcc_0, r8, SUCCESS) [post-build]
```

The predicate on line 2 is the same as before, but has been marked as a *build* predicate. The new predicate on line 3 describes an alternative. With post-build checks, a command is only changed if its predicates fail in both the *build* and *post-build* scenarios (line 10 of EVALTRACE). In other words, a command is changed only when its dependencies are distinct from both the dependencies observed during the last build and immediately after the last build.

5.1 Command Emulation and Execution

RIKER begins every build by emulating a *root command*. The root command sets up initial references (e.g. the root directory, working directory, standard streams, etc.) and then launches `Rikerfile`. When an emulated command contains a LAUNCH statement, RIKER will either emulate or execute the child command depending on whether or not the child is in R . If the child is in R RIKER will launch the command in a new process with system call tracing. All statements from the child command are discarded from the transcript, and will be replaced with new TraceIR collected from the child's execution. If c is not in R , RIKER simply emulates the child from the build transcript.

Parent commands typically wait for their children to exit using the `wait` system call; this system call generates a JOIN statement in the build transcript. RIKER emulates a JOIN statement by handling traced system calls from build processes until the child command's main process exits.

When a command c is executed it may depend on artifacts modified by emulated commands. The latest state of these artifacts is in M , not on the actual filesystem. RIKER will write these changes out to the filesystem as they are needed, either when c begins execution (when file descriptors are inherited by the child) or when c first accesses the artifact during its execution. This mechanism is critical for RIKER's ability to incrementalize builds, and relies heavily on caching.

5.2 Caching

Without caching, RIKER's ability to execute sub-commands in isolation would be limited because many of the needed inputs would not be available. `gcc` and other language tools routinely create "ephemeral" state—like temporary files—

communication channels for tools in the toolchain. RIKER's caching and TraceIR make it possible to automatically restore this ephemeral state to run a command whose inputs are produced by other commands (e.g. the assembler or linker).

The motivating example in Figure 1 illustrates this functionality. Suppose a user edits `main.c`; RIKER will execute the compiler and assembler to produce `tmp2.o`, but does not need to execute any commands to produce the other `.o` files. Instead, these files are simply restored from cache when they are first accessed by the linker. The fact that `gcc` reuses `tmp.s` is not a problem, as each use of the file is ordered in the build transcript so RIKER always knows which version of the file is required for every command.

RIKER caches files, symlinks, and directories. Cached artifacts are stored in a `.rkr` directory, and are garbage collected when RIKER detects that they are no longer referenced by the build transcript. RIKER currently does not cache pipes, sockets, or special files. If a command that reads from a pipe must run, RIKER will also run commands that write to that pipe to provide uncached inputs.

5.3 Build Planning

The purpose of build planning (line 7 in DOBUILD of Figure 2) is twofold: to ensure the build terminates, and to improve efficiency. PLAN works much like the mark-sweep garbage collection algorithm [18] and uses the command dependence graph, D , returned by EVALTRACE. D is a digraph of producer-consumer relationships between commands.

Commands are marked under a few conditions: a) a command is in R , having directly observed a change in EVALTRACE; b) a command consumes uncached input produced by a command already marked to run; c) a command produces uncached output consumed by another command already marked to run; or d) a command produces uncached output that should persist after the build.

The above criteria identify commands that subsequent build phases in a cycle-free build would eventually identify, so for those builds it reduces the number of phases. However, in builds with dependence cycles, RIKER may not terminate without special handling. In D , a cycle appears as a strongly-connected component (SCC) [31]. Planning ensures that commands in a cycle run atomically. Caching also breaks cycles; RIKER only needs to atomically run SCCs that interact through uncached artifacts like pipes.

5.4 Exit Code Handling

Unlike prior forward build systems, RIKER can execute a sub-command without executing its parent. Parent commands can observe the exit codes of their children, so if a child finishes with a different exit code the parent must run. Because exit code changes are rare, RIKER optimistically assumes that the child's exit code will not change. In the common case the child command runs, finishes with the expected exit code, and the build is complete. If the child finishes with a different

exit code, the parent observes a change and will re-execute in the next phase of the build. Executing the parent may re-execute the child if the child's dependencies change again. In the worst case, RIKER could backtrack on every command, taking $O(n^2)$ time, where n is the number of commands. Even for builds that contain compilation errors—and thus changed exit codes—we observed that total work done is close to $O(n)$.

5.5 Correctness

Here we provide a proof sketch for the correctness of RIKER's incremental build algorithm. We make the following assumptions.

- A1. The user-provided full build specification does what the user intends.
- A2. The user accepts that equivalent outputs (defined in §4.1) are interchangeable, an assumption shared by most build systems.
- A3. Intercepting system calls is sufficient to determine all dependencies.
- A4. Our translation from system calls to TraceIR faithfully captures dependencies and side-effects. Our empirical evaluation in Section 6 provides evidence that this translation is accurate.

An incremental build tool that produces outputs that could not have come from a full build is clearly incorrect. A *consistent* build produces output that could have come from a full build, and is therefore correct (A1, A2) [12, 13].

Running an empty build specification produces no output, so all rebuilds of this specification are by definition consistent. Given a consistent build with k commands, add command $k + 1$ to the specification. Command $k + 1$ may depend on outputs from any of the preceding k commands. The build remains consistent when $k + 1$ is executed (A1). On rebuild, if $k + 1$ is unchanged, skipping command $k + 1$ preserves consistency because RIKER restores cached artifacts (A2). By induction, a build is consistent as long as all changed commands are executed.

RIKER executes changed commands in phases. If an executed command produces a different output read by another command, the latter command is changed and will execute in the next phase. The build terminates when a phase finishes with no changed commands. Since RIKER's algorithm executes all changed commands, it produces consistent builds.

The proof sketch above assumes commands k and j do not participate in a dependence cycle, but these cycles arise in real builds. Without loss of generality, assume k writes output that j accesses, and later j writes output that k accesses; RIKER's build transcript captures the temporal order of these interactions. We allow for such cycles by logically partitioning k into k' , the portion of k that runs before its dependence on j , and k'' , the remainder of k . Now j depends on k' and k'' depends on j , so there is no longer a cycle. We add the constraint that if either k' or k'' must run, both will run, as these are actually two parts of the same command.

6 Evaluation

Our evaluation of RIKER addresses four key questions:

- Q1: Are RIKER builds easy to specify?
- Q2: Are RIKER builds fast?
- Q3: Are RIKER builds correct?
- Q4: How does RIKER compare to RATTLE?

We use RIKER to build 14 software packages, including large projects like LLVM, memcached, redis, and protobuf. Evaluation was conducted on a typical developer workstation with an Intel Core i5-7600 processor, 8GB of RAM, and an SSD running Ubuntu 20.04 with kernel version 5.4.0-80. Builds use either gcc version 9.3.0 or clang 10.0.0.

6.1 Are RIKER builds easy to specify?

To answer this question, we wrote *Rikerfiles* for seven applications: lua, memcached, redis, rkr, sqlite, vim, and xz. The new builds produce the same targets as the projects' existing make or cmake builds. Unlike the default build systems, the RIKER-based builds do not list any dependencies or incremental build steps. Three of these builds were written by undergraduate students over the course of a few days; the students were new to RIKER and unfamiliar with the project sources they were building. The biggest challenge the students faced was understanding the existing build specifications, a task that is likely easier for the project's own developers.

A key feature of a *Rikerfile* is its brevity, illustrated by memcached's complete *Rikerfile*:

```
1 CFLAGS="..."
2 DEBUG_CFLAGS="..."
3 MEMCACHED_SRC="memcached.c hash.c ..."
4 TESTAPP_SRC="testapp.c util.c ..."
5 gcc $CFLAGS -o memcached $MEMCACHED_SRC -levent
6 gcc $DEBUG_CFLAGS -o memcached-debug \
7   $MEMCACHED_SRC -levent
8 gcc $CFLAGS -o sizes sizes.c -levent
9 gcc $CFLAGS -o testapp $TESTAPP_SRC -levent
10 gcc $CFLAGS -o timedrun timedrun.c -levent
```

This level of simplification is typical; Our largest *Rikerfile*—used to build sqlite—is just over 5KB, and consists mostly of a list of source files. Forward builds are easier to specify because of automatic dependency discovery. RIKER's incremental builds are also significantly shorter than specifications for prior forward build tools (see §6.4).

6.2 Are RIKER builds fast?

The first full build of any software project is usually the longest build. Full builds are where RIKER incurs the largest absolute overhead. Importantly, full builds are not the common case; developers run incremental builds far more often than full builds. This section shows that even with the extra delay, full builds are reasonably fast with RIKER.

To measure RIKER's overhead, we built 14 software projects with RIKER. Seven of these projects use a *Rikerfile* that replaces the default build (see §6.1), while

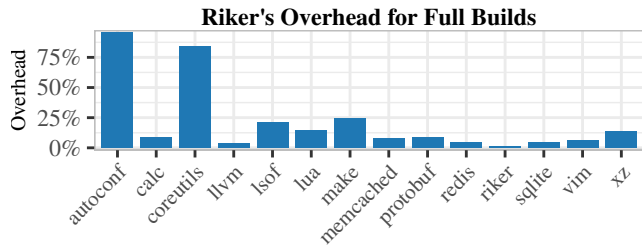


Figure 3: RIKER runtime overhead for full builds compared to each project’s default build system. RIKER’s median overhead on full builds is 8.8%, with a median absolute slowdown of 1.2s.

the other half use a `Rikerfile` that *wraps* the default build. The only requirement for a `Rikerfile` is that it run a full build, so a `make`-based project can be built with a one-line `Rikerfile`: `make --always-make`. Unfortunately, tracing `make` itself can lead to spurious dependencies so this approach is only suitable for evaluating full-build performance.

Figure 3 shows the results of running full builds with RIKER. These builds are run in serial; we examine the performance of parallel builds later in this section. Each project is built five times with RIKER and its default build system, with the exception of LLVM which we build three times due to its long build time. Median full-build overhead for all benchmarks is just 8.8%; most builds have between 4% and 20% overhead. TraceIR transcript sizes are roughly proportional to build time, ranging from 2MB for `autoconf` (1.2s build) to 264MB for LLVM (77-minute build). In absolute terms, RIKER spends a median of just 1.2 seconds longer to perform a full build than each project’s default build system. The longest additional waiting time for a RIKER build is for LLVM, which takes about three minutes longer than the default 77 minute build (4% overhead). The worst overheads appear in projects like `autoconf` and `coreutils` that build many small programs rather than a single large executable. RIKER’s full-build overhead is less than 25% for all other projects.

Incremental Builds. The most important measure of efficiency for a build system is its ability to perform fast incremental rebuilds. We perform two experiments to measure the efficiency of RIKER’s incremental builds.

First, we measure the time it takes RIKER to perform a *no-op build*—one where no commands need to run—by running an incremental build immediately after finishing a full build. The median RIKER no-op build time over the 14 benchmarks is just 220ms, compared to 5ms for the default build system. The longest additional wait is for the LLVM build, which takes 11.3s with RIKER compared to 4.8s with `make`. More than half of the no-op builds with RIKER take just 162ms longer than the default build system, an imperceptible difference.

Second, we use real developer commits to measure the efficiency of performing incremental builds with RIKER versus a project’s default build system. We run this experiment on six projects—`memcached`, `redis`, `rkr`, `sqlite`, `vim`, and `xz`—all of which have custom `Rikerfiles` and public version control

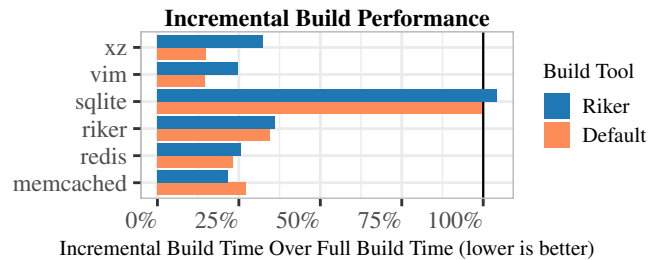


Figure 4: Time to run 100 incremental builds as a percentage of the time to run a full build at every commit using the project’s default build system. Excluding `sqlite`, RIKER’s incremental builds save 235 minutes compared to 250 minutes saved by the default build system.

histories. We perform a full build of each project, and then measure the time required to update the build at each of the next 100 commits in the project’s `git` repository. This experiment simulates a developer performing incremental builds after editing a subset of the project’s source files.

Figure 4 shows the results of this second experiment. The graph shows the total time required for all 100 incremental builds as a percentage of the time it would take to run a full build at each commit. Note that we compare RIKER’s incremental build times to the time it would take to run a full build with the project’s *default* build system. This ensures RIKER’s overhead on the full build does not give it an advantage compared to the slightly faster default build system.

95% of RIKER’s incremental builds complete within 3.8s of the default build system. In every benchmark except one (`sqlite`), RIKER is able to reduce build time by at least 63% relative to a full build. Over 5000 incremental builds of these five benchmarks, the default build system reduces build time by 74.7%, saving 250 minutes compared to full builds at each commit. RIKER saves 70.1% of total build time—a total of 235 minutes. Building with RIKER yields 94% of the benefit of a manually-specified incremental build, but with *no manual effort* and *no risk of errors*.

RIKER is able to save more time than the default build system for `memcached` because the case study includes several commits that edit the build specification itself. These edits generally require a full build for `make`-based projects, but RIKER is still able to perform a safe incremental build when the specification changes.

Neither RIKER nor `make` saves any work when building `sqlite`. This is because `sqlite`’s build concatenates all of its source files together before compiling them so the project can be distributed as a single source file. RIKER’s overhead adds less than three seconds (4.2%) to each “incremental” build.

Parallel Builds. Our evaluation has so far focused on serial builds, but parallel builds are also a useful mechanism for reducing build times. There are two concerns when it comes to RIKER’s support for parallel builds. First is scalability: how does RIKER’s tracing impact the performance of parallel builds? The second concern is expressiveness: how do users write a `Rikerfile` that describes a parallel build?

To assess RIKER’s scalability we focus on LLVM, our largest benchmark. On our evaluation machine, which has four cores, we see good scalability when adding up to four parallel jobs to LLVM’s `make` build. Build time is reduced by 54.3%, 64.4%, and 71.9% when building with two, three, and four workers respectively. With RIKER, we see reductions in build time of 47.7%, 63.0%, and 67.0% for the same numbers of workers. RIKER’s overhead increases from 4.0% for the serial build to 22.2% with four workers. A likely cause for this reduction in scalability is that RIKER utilizes a busy-wait loop that monopolizes a core. Even with this limitation—which we plan to address—parallel builds with RIKER are still significantly faster than serial builds.

While the prior experiment examines RIKER’s tracing performance on a parallel `make` build, it is also possible to write parallel build specifications directly in a `Rikerfile`. To make this as easy as possible, RIKER includes a wrapper around common C/C++ compilers that launches sub-commands for compilation in parallel, which is always safe (two `.c` files can always be compiled simultaneously). With RIKER’s compiler wrapper, simple lines like `gcc *.c` start parallel compilation tasks for each source file, which RIKER can also run in parallel for later rebuilds. Enabling this wrapper reduces build time for `memcached` by about 50%, compared to 60% with `memcached`’s own parallel `make` build. RIKER’s compiler wrapper provides an easy, automatic way to run parallel builds, and RIKER’s tracing does not significantly limit scalability.

6.3 Are RIKER builds correct?

We run each project’s full test suite for both the default and RIKER builds. For the six projects in the previous section, the tested outputs are the product of one full build and 100 incremental builds, one for each commit. The remaining projects run only full builds. Every RIKER project passes exactly the same tests as the original build system. This experiment provides evidence that RIKER correctly updates builds to produce equivalent final targets.

We have additional confidence that RIKER’s translation from system calls to TraceIR and its POSIX filesystem model are correct because RIKER checks the outcomes of operations in the model against actual system call results. RIKER raises a warning if the model deviates from actual system behavior; our experiments and test suite raise no such warnings.

6.4 How does RIKER compare to RATTLE?

To compare against the prior state of the art forward build system, we ported the `memcached` build to RATTLE [23]. Our efforts to port other benchmarks to RATTLE were not successful. RATTLE limits state modeling to files, meaning RATTLE misses some kinds of changes [29, 30]. We demonstrate with the following RATTLE build:

```
main :: IO ()
main = rattleRun rattleOptions $ do
  cmd "gcc prog.c"
```

```
cmd "mkdir dir"
cmd "mv a.out dir"
```

The full build runs correctly. However, because RATTLE does not model directories, changing `prog.c` leads to an inconsistent rebuild. A new `a.out` is placed in the current directory, leaving the original in `dir` untouched. RATTLE cannot build `sqlite` for precisely this reason. RATTLE also does not handle circular dependencies. The `lua` build calls `ranlib`, which is used to create library archives. Since `ranlib` modifies its input, RATTLE fails during the full build.

We were able to build `memcached` with RATTLE, which imposes a median overhead of less than 1% for the full build. This is because RATTLE uses library interposition for tracing, which is faster than `ptrace` but will miss system calls not issued by `libc`. A straightforward translation of the build specification from RIKER to RATTLE does *not* result in good incremental build performance; RATTLE cannot run fine-grained incremental builds from simple specifications, so it only reduces build time by 25% compared to full builds over the 100 commits from our earlier experiment. This is significantly less than the 78% build time reduction RIKER is able to achieve from the simple build specification. A RATTLE specification with comparable incremental build performance requires 63 separate commands, compared to just five for RIKER.

7 Conclusion

RIKER significantly lowers the burden of correctly specifying fast incremental builds. A RIKER build specification can be any executable program, like a simple shell script that performs a full build. In many cases, even a single build command such as `gcc *.c` is sufficient. Users do not need to list dependencies in their build specifications, nor are they required to break builds into incremental steps. Nevertheless, RIKER always builds correctly and quickly.

RIKER uses low-overhead system call tracing to automatically discover dependencies as build commands execute, and on rebuild, runs only the subset of commands required to bring the build up-to-date. RIKER has a median overhead of 8.8% across 14 real software packages, and it realizes 94% of `make`’s incremental build speedup with no manual effort and no risk of errors. We think these substantial engineering improvements are well worth RIKER’s modest overheads. RIKER is available under an open-source license at <https://rkr.sh>.

Acknowledgments

This work was supported by National Science Foundation grants CNS-2008487 and CNS-2008940. We thank Michael Chovanak, Yolanda Jiang, Alissa Johnson, Philip Ma, Abigail Munsen, Jonathan Sadun, and Linh Tang, who contributed to RIKER’s implementation while they were students at Grinnell College. We also thank Emery Berger, John Vilks, and Benjamin Zorn for their feedback and guidance.

References

- [1] The Ninja Build System. <https://ninja-build.org/>, November 2020. v1.11.0.
- [2] Apache Software Foundation. Apache Ant. <https://ant.apache.org>, October 2021. v1.10.12.
- [3] Apache Software Foundation. Apache Maven. <https://maven.apache.org/>, March 2022. v3.8.5.
- [4] CMake Project. CMake. <https://cmake.org>, May 2022. v3.23.2.
- [5] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. A Sound and Optimal Incremental Build System with Dynamic Dependencies. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015*, pages 89–106, New York, NY, USA, 2015. ACM.
- [6] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. CloudBuild: Microsoft’s Distributed and Caching Build Service. In *Proceedings of the 38th International Conference on Software Engineering Companion, ICSE ’16*, page 11–20, New York, NY, USA, 2016. Association for Computing Machinery.
- [7] Facebook. Buck: A high-performance build tool. <https://buck.build/>, January 2021. v2022.05.05.01.
- [8] Stuart I. Feldman. Make — a program for maintaining computer programs. *Software: Practice and Experience*, 9(4):255–265, 1979.
- [9] GNU Project. GNU Autoconf. <https://www.gnu.org/software/autoconf/>, January 2021. v2.71.
- [10] GNU Project. GNU Automake. <https://www.gnu.org/software/automake/>, October 2021. v1.16.5.
- [11] Google. Bazel. <https://bazel.build/>, June 2022. v5.2.0.
- [12] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. SRC Technical Note 1999-001: The Vesta Approach to Software Configuration Management. *Compaq Systems Research Center*, June 1999.
- [13] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. *Software Configuration Management System Using Vesta*. Springer Science & Business Media, 2006.
- [14] Ben Hoyt and Simon Alford. Fabricate. <https://github.com/brushtechnology/fabricate>, October 2020.
- [15] Leslie Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System*, page 179–196. *Concurrency: The Works of Leslie Lamport*. Association for Computing Machinery, New York, NY, USA, 2019.
- [16] Linux man-pages project. *path_resolution(7)*, *Linux User’s Manual*, November 2017.
- [17] Linux man-pages project. *ptrace(2)*, *Linux User’s Manual*, March 2021.
- [18] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960.
- [19] Bill McCloskey. Memoize: A replacement for make. <http://www.eecs.berkeley.edu/~billm/memoize.html>, June 2008. Archived: 2010-09-05.
- [20] M.D. McIlroy, E. N. Pinson, and B. A. Tague. Unix Time-Sharing System: Forward. *The Bell System Technical Journal*, 57(6):1899–1904, 1978.
- [21] Peter Miller. Recursive Make Considered Harmful. *Journal of AUUG Inc.*, 19(1), March 1997.
- [22] Neil Mitchell. Shake Before Building: Replacing Make with Haskell. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming, ICFP ’12*, pages 55–66, New York, NY, USA, 2012. ACM.
- [23] Neil Mitchell. Rattle. <https://github.com/ndmitchell/rattle>, May 2020. v0.2.
- [24] Robert O’Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering Record and Replay for Deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 377–389, Santa Clara, CA, July 2017. USENIX Association.
- [25] The Linux Kernel Project. Seccomp BPF (SECure COMputing with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.
- [26] SCons Foundation. Scons. <https://scons.org>, February 2022. v4.3.0.
- [27] Mike Shal. Tup Build System. <https://gittup.org/tup/>, May 2021. v0.7.11.
- [28] Thodoris Sotiropoulos, Stefanos Chaliasos, Dimitris Mitropoulos, and Diomidis Spinellis. A Model for Detecting Faults in Build Specifications. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), November 2020.

- [29] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. Build Scripts with Perfect Dependencies. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), November 2020.
- [30] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. Forward Build Systems, Formally. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 130–142, New York, NY, USA, 2022. Association for Computing Machinery.
- [31] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

A Artifact Appendix

Abstract

The RIKER artifact includes a virtual machine image with a pre-installed copy of RIKER. RIKER is a build system that automatically discovers and runs incremental builds from simple specifications. This artifact makes it easy to reproduce the experiments described in the paper, which evaluate RIKER’s performance and effectiveness as a build tool.

Scope

This artifact should be used for two purposes: a) to reproduce the experiments from the paper, and b) to generate plots from the new results. The provided scripts reproduce the full-build overhead plot (Figure 3), the incremental savings plot (Figure 4), and the summary statistics reported in the abstract and in Section 6. We expect runtime overhead numbers will vary across platforms. Running the evaluation within a virtual machine will also likely have some effect on overhead. However, the artifact’s overhead should be close to the paper’s reported 8.8% median overhead on full builds, and incremental savings should be close to `make`’s incremental build performance.

Contents

README: A detailed guide to setting up the artifact and running experiments from the paper.

Virtual Machine Image: A VM image in OVA format that contains RIKER’s source code, build dependencies, and scripts that automatically run the paper’s benchmarks.

Hosting

The artifact is available at <https://doi.org/10.5281/zenodo.6544966>. Updated versions of RIKER will be available at <https://rkr.sh>. We recommend using the updated version for uses other than reproducing experimental results from the paper. Newer versions are likely to be more stable, support more kernel versions and architectures, and include bug fixes and additional features.

Requirements

Hardware Requirements. The virtual machine included with this artifact requires an x86_64 machine.

Software Requirements. The virtual machine image includes all build and evaluation dependencies. The OVA file can be imported into any hypervisor that supports OVA, but it was developed and tested using VirtualBox. The experimental evaluation requires network access; other hypervisors may require changes to network configuration after import.



FlatFS: Flatten Hierarchical File System Namespace on Non-volatile Memories

Miao Cai^{†‡§}, Junru Shen[‡], Bin Tang^{†‡}, Hao Huang[§], Baoliu Ye^{§†‡}

Key Laboratory of Water Big Data Technology of Ministry of Water Resources, Hohai University[†]

School of Computer and Information, Hohai University[‡]

State Key Laboratory for Novel Software Technology, Nanjing University[§]

Abstract

The conventional file system provides a hierarchical namespace by structuring it as a directory tree. Tree-based namespace structure leads to inefficient file path walk and expensive namespace tree traversal, underutilizing ultra-low access latency and good sequential performance provided by non-volatile memory systems. This paper proposes FlatFS, a NVM file system that features a flat namespace architecture while provides a compatible hierarchical namespace view. FlatFS incorporates three novel techniques: coordinated file path walk model, range-optimized NVM-friendly B^r tree, and write-optimized compressed index key layout, to fully exploit flat namespace structure to improve file system namespace performance on high-performance NVMs. Evaluation results demonstrate that FlatFS achieves significant performance improvements for metadata-intensive benchmarks and real-world applications compared to other file systems.

1 Introduction

Large, deep directory tree stresses file system namespace performance. For example, Hive, a famous data warehousing software system, uses a column-based partition technique to provide efficient query for big database tables [39]. Every column value represents a path component, and all columns constitute a file path to index the data. Such a partition technique causes a large deep directory tree for a wide table, making data query prohibitively expensive.

The recent advent of ultra-fast non-volatile memories revolutionized file system design, shifting performance bottlenecks from hardware I/O to the software stack. However, the current file system namespace structure, which is designed for slow storage devices, encounters severe performance issues with high-performance non-volatile memory systems [1, 20, 30, 36].

The file system provides a hierarchical namespace that is structured as a directory tree [5, 37], as shown in Figure 1a. The virtual file system (VFS) unifies multiple volatile namespace hierarchies to provide a single global namespace

view [40]. For any metadata system calls, the VFS first performs a pathname lookup (i.e., path walk) by walking the directory tree to find the associated directory entry (dentry), and then performing metadata operations, e.g., changing file permissions. Hierarchical namespace structure causes two major performance problems: *inefficient file path walk* and *expensive namespace tree traversal*.

First, current file path walk is slow and non-scalable. Resolving a path component involves costly dentry search and other coupled system operations like security module enforcement [10, 23, 24, 40]. Moreover, suppose a file path contains n components, resolving the last component needs to locate previous $n - 1$ components. The total path walk task is linear to the number of components in the file path. The path walk efficiency has a slight performance impact on system calls for slow storage devices since hardware I/O dominates. In contrast, for NVMs offering ultra-low access latency [17, 42, 46], such high critical-path latency is unacceptable (see §2.1).

Second, traversing the namespace tree recursively is expensive with the hierarchical namespace structure. Entries of different directories are physically scattered over the storage device. It results in poor data access locality and indirect memory addressing during tree traversal. Namespace tree traversal is prevalent in common system usages such as *find*, *rm -r* and *ls -R*. Moreover, commercial NVMs like Intel Optane memory provide $> 5GB/s$ memory bandwidth and prefer sequential memory access behaviors [17, 46]. Unfortunately, hierarchical namespace structure fails to utilize this important system characteristic (see §2.2).

Research efforts have been devoted to addressing these two problems. Tsai et al. [40] propose full-path caching to reduce path walk latency in the VFS layer. Solaris also incorporates a similar path-to-vnode cache [23]. However, caching is heavily dependent on file access locality. ByVFS [41] bypasses the VFS directory tree and directly manipulates dentry from the file system. As NVM devices deliver unbalanced read/write performance, the dentry write performance is sub-optimal. A file system level optimization is utilizing efficient data structures (e.g., radix tree [44], hash table [15, 16], B^+

tree [5, 34, 37], skip list [13]) or key-value stores (e.g., LevelDB [2, 32] and TokuDB [6, 18]) to manage and index persistent directory entries. Though these data structures offer good indexing performance, the data locality problem is not well-addressed. Moreover, all existing in-kernel file systems are developed underneath the VFS framework [14, 18, 26, 31, 44, 48]. Therefore, pathname lookup performance in these file systems is constrained by VFS path walk efficiency.

This paper revisits namespace structure for ultra-fast, byte-addressable NVMs and proposes a novel file system named FlatFS. FlatFS exhibits a flat namespace architecture but still provides a compatible hierarchical namespace view. File metadata are directly indexed by their unique hierarchical paths. Hierarchical paths are organized without any further structure.

The flat namespace structure brings two performance advantages for file system design for NVMs. First, file path walk is fast and scalable as it only requires a single pathname lookup to flat namespace irrespective of path length. It effectively shortens the path walk for metadata system calls. Second, data locality is improved because contiguous directory entries in the namespace are also stored consecutively on the storage device. It accelerates file system namespace traversal and benefits directory range operations.

To exploit flat structure to improve namespace operation performance on high-performance NVMs, FlatFS incorporates three core techniques. First, a coordinated path walk is proposed to orchestrate two distinct path walk models in a global unified VFS namespace §4.1. FlatFS achieves fast, scalable path walk by separating pathname lookup from other intricate system operations, yet preserves the system semantics as the conventional path walk model. Instead of reconstructing the existing path walk module, coordinated path walk offers a flexible and backward-compatible solution to integrate a distinctive path walk model into the global namespace.

Second, a range-optimized NVM-friendly B^+ tree is devised to manage variable-sized index keys §4.2. B^+ tree provides efficient data-structure-level range operations (e.g., *range insert*) in logarithmic time. It effectively remedies the directory move shortcoming for the flat namespace, as well as facilitates designing other fast directory range operations §4.3.

Third, a write-optimized compressed (WoC) index key design is proposed by leveraging NVM byte-addressability to improve variable-sized key storage efficiency as well as reduce expensive small memory writes and data persistence overheads in key management on NVM systems §4.4.

Finally, FlatFS achieves low-cost metadata crash-consistency for the flat namespace §4.5. The WoC key design effectively reduces the performance costs for data consistency in index key insert and removal. For directory range operations that involve complex tree structure manipulation, FlatFS also simplifies namespace tree crash-safety design based on an insight derived from B^+ tree structure primitives.

We evaluate FlatFS with extensive experiments using both benchmarks and three applications (a version control system

Git, a parallel file indexer Psearchy [11], and a data warehouse software Hive [39]) on Intel Optane DC Persistent Memory §5. Evaluation results demonstrate that our flat namespace fully unleashes the high-performance NVM system. FlatFS outperforms other file systems by a factor of 4.02× for micro- and macro-benchmarks and improves metadata-intensive application performance by up to 37.5%.

In summary, this paper makes the following contributions:

- We analyze two performance issues in hierarchical namespace structure with high-performance NVMs.
- We propose a metadata-optimized file system FlatFS with three core techniques to flatten the hierarchical namespace for fast, byte-addressable NVM systems.
- We conduct extensive experiments to demonstrate the performance benefits of FlatFS to both metadata-intensive benchmarks and applications.

2 Background and Motivation

This section takes Linux kernel as an example to describe performance issues in the current file system namespace.

2.1 Inefficient File Path Walk

Costly component resolution. We conduct an experiment to demonstrate path component resolution inefficiency. We measure the execution latencies of six typical metadata system calls (*creat*, *open*, *stat*, *chmod*, *unlink*, *mkdir*) in NOVA file system on our testbed machine. Every system call operates on a file in a six-depth directory. Figure 1b shows the system call execution time and performance breakdown.

When the dentry cache (dcache) is hot, profiling results show that an average 15.87% execution time is spent on the dcache lookup. The dcache lookup includes filename hashing and hash table lookup. Currently, there is only a centralized dcache in the VFS layer. Increasing dentries stresses the dcache indexing performance. The permission checking takes 8.83% of the execution time on average. The path walk also spends 11.65% execution time on other system operations (denoted as *Other Walk* in Figure 1b), such as mount point checking and reference counter updating. In summary, when the dcache is hot, resolving a six-component file path occupies 14.17%-67.19% execution time for five system calls. The file operations take an average of 31.49% execution time.

On the other hand, when the dcache is cold, the underlying file system has to search the missing dentry. Dentry lookup of NOVA file system takes 69.26% of execution time for cold dcache. The dentry miss penalty is high. Besides the I/O transfer for the missing dentry, it also includes searching the dentry in the storage device and inserting this dentry into various VFS management data structures. Table 1 compares the block reading and dentry lookup latencies on NVM and SSD. Relatively slow SSD delivers long I/O latency, which dominates the overall execution latency. However, the dentry lookup time percentage increases around 20% when the storage device shifts from SSD to NVM. It indicates that

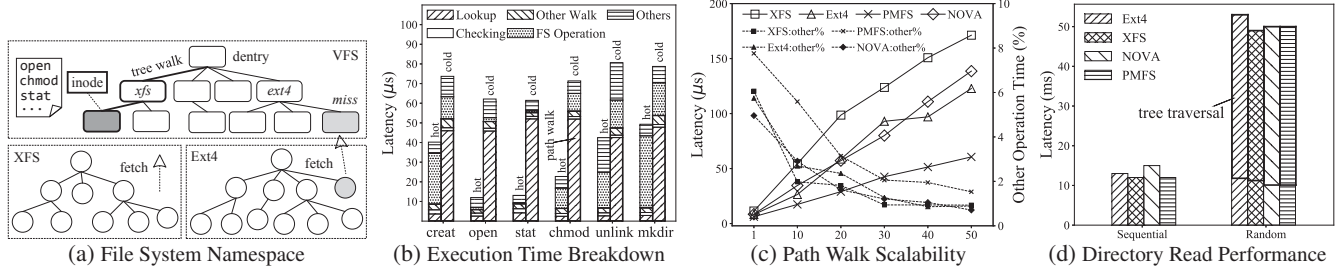


Figure 1: File System Namespace Structure in Linux and Performance Issues

Table 1: Comparing Block Read and Dentry Lookup Latency on NVM and SSD

	SSD: block reading (%)	SSD: dentry lookup (%)	NVM: block reading (%)	NVM: dentry lookup (%)
Ext4	58.74%	4.12%	27.11%	25.36%
XFS	60.98%	3.34%	29.45%	28.10%

the performance bottleneck transfers from the hardware I/O latency to software path walk design with ultra-fast NVMs.

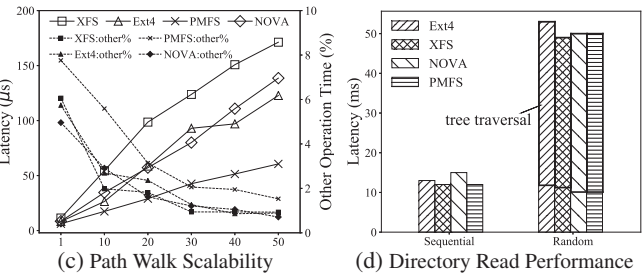
Non-scalable path walk. The current component-at-a-time path walk design is non-scalable with the number of path components. Such path walk design is widely adopted in current operating systems as it is convenient to implement the component resolution. Moreover, many other system functionalities are implemented upon component resolution [10, 24]. They further increase the latency and exacerbate the path walk scalability issue. In addition, it is difficult to decouple them from the path walk.

We perform an experiment to understand the path walk scalability problem. We measure the *stat* syscall latency with different path component numbers on four NVM file systems. The VFS dcache is cold in the experiment. As shown in Figure 1c, the *stat* execution latency of four file systems increases dramatically as path length increases. The operation latency of the 50-component file path is nearly 14× higher than the 1-component file path. The VFS invokes file system-specific *getattr* to retrieve the file attributes. However, *getattr* only takes 1.01% on average of four file systems for a 50-component path configuration (denoted as *FS:other%* in Figure 1c). The rest of the execution time is spent on resolving the lengthy file path.

Summary. Current slow, non-scalable file path walk design has a large impact on metadata system call performance. This is a common problem for all file systems running on different devices like SSDs or HDDs. However, as commercial NVM devices offer near-DRAM access latency, the major bottleneck shifts from long hardware I/O to software path walk design, motivating us to reduce such software latency.

2.2 Expensive Namespace Tree Traversal

Recent Intel Optane memory studies report that there is an asymmetry between sequential and random access performance [17, 28, 43, 46]. The performance gap between sequential and random memory bandwidth ranges from 2.3× to 3.5×. Moreover, the memory access latency is also sensitive to sequential and random access patterns [17, 42, 46].



File systems directly persist their directory tree in the NVM device. Traversing such a hierarchical namespace tree recursively is expensive. The reason is twofold. First, as directories logically form a tree structure, traversing the hierarchical tree causes indirect memory accesses. Second, persistent tree traversal introduces random memory access as dentries of different directories are distributed across the device. Previous studies show that both these memory access behaviors are suboptimal [17, 46]. Recursive tree traversal is an important namespace operation and used by many real-world applications heavily (e.g., *cp -r*, *git status*). Moreover, as page cache is removed from the NVM storage stack [26, 31, 44, 48], directory reading operations are directly performed on the NVM device. Existing persistent namespace tree traversal degrades directory reading performance.

We perform an experiment to understand the performance costs of hierarchical namespace tree traversal. We create two sets of 1024 files. Files in the first set are stored under an eleven-depth directory hierarchy. All files of the second set are stored in a one-depth directory. We use *ls -R* to read these two directories recursively. The first listing operation walks a hierarchical namespace and the second listing operation simulates sequential access in a flat namespace. Figure 1d shows that sequential directory reading performs 4.08× better than random directory reading. Traversing the hierarchical directory tree occupies nearly 80% total execution time.

Summary. The hierarchical namespace structure leads to low directory reading performance due to expensive tree traversal. As the page cache is removed from the modern NVM storage stack, directory reading operations access data stored on NVM devices directly. How to architect the file system namespace structure to exploit the device characteristic to improve the namespace operation performance still remains unsolved.

3 Overview

3.1 Design Goals

Through flattening the hierarchical namespace, we build a high-performance, POSIX-compliant NVM file system. In particular, we have four design goals.

- **Short path walk.** File path walk is essential for most of the metadata system calls. FlatFS aims to reduce such software latency to minimize performance impacts on system call execution.

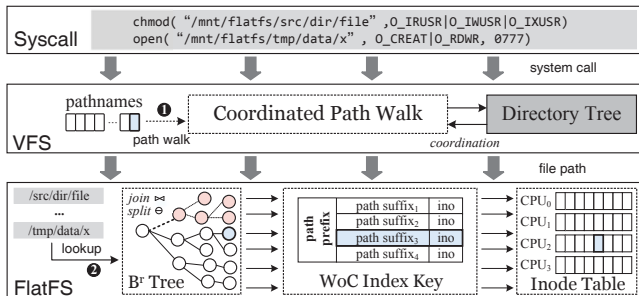


Figure 2: FlatFS System Architecture

- **Optimizing range operation.** As sequential accesses are optimal in NVMs, FlatFS aims to re-structure namespace architecture to fully exploit this system characteristic.
- **Reducing persistence costs.** FlatFS applies write-optimization techniques to key layout design to reduce data persistence costs in its index key management.
- **Ensuring namespace crash-safety.** FlatFS achieves namespace crash-safety guarantees for conventional metadata syscalls and compound directory range operations.

3.2 FlatFS Architecture

FlatFS system architecture is presented in Figure 2. There is no cached namespace in the VFS. The namespace operation is directly performed on the device. There are four types of files in FlatFS: regular file, directory, symbolic link, and hard link. The file metadata is indexed by its full pathname relative to the FlatFS mount point.

In the system call layer, FlatFS provides a compatible hierarchical namespace view. Applications still use hierarchical file paths to access files and directories in FlatFS namespace.

In the virtual file system layer, we design a coordinated path walk (§4.1). The FlatFS namespace also appears in the VFS namespace to preserve a global unified namespace view. The coordinated path walk incorporates two distinct path walk models. It cooperates with the VFS directory tree to perform path walk across distinctive file system namespaces and dispatches requests to the corresponding file system instance.

In the file system layer, index keys are managed by a range-optimized persistent B^+ tree (§4.2). All index keys are sorted in an lexicographical order. B^+ tree is carefully designed with NVM properties [46]. Besides, B^+ tree supports data structure level range operations based on two proposed tree structure primitives: *join* and *split*. It facilitates designing low-cost directory range operations (e.g., *rename*) and simplifies their implementation (§4.3).

In the key storage layer, the idiosyncratic NVM systems pose severe challenges to index key management. Frequent index key updates incur a large number of small random memory writes and cache line flushes, which is especially harmful to system performance [46]. FlatFS adopts a write-optimized compressed key layout to avoid most memory writes and cache line flushes during key inserts and removes (§4.4).

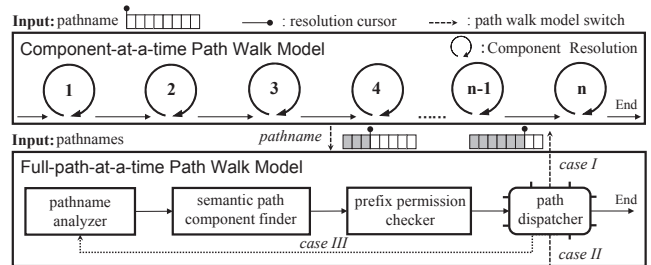


Figure 3: Two Path Walk Models

4 Design and Implementation

4.1 Coordinated File Path Walk

Coordinated file path walk applies two path walk models to resolve a pathname. A path component could be in five different forms: “.” (dot), “..” (dot-dot), normal file, directory, and symbolic link (symlink). Figure 3 illustrates these two path walk models. The component-at-a-time path walk model resolves path component one by one while the full-path-at-a-time path walk processes the whole pathname at a time. This section describes how our path walk model correctly handle different kinds of path components and permission checking as well as coordination between these two models.

Pathname analyzer. The pathname analyzer generates a canonical pathname without any dots and redundant slashes by performing lexical processing on the file path. If FlatFS adopts the Plan 9 lexical file pathname [29], the analyzer resolves a dot-dot component by removing the path component before it. Otherwise, the dot-dot component is handled by the semantic path component finder, which will be described later. After analyzing, FlatFS passes the canonical pathname to the semantic path component finder.

Semantic path component finder. The pathname analyzer can only handle non-semantic path components. The semantic components (i.e., symbolic links and mount points) are identified by the semantic path component finder using a key indexing approach. Specifically, these two kinds of semantic components have associated entries $\langle \text{path}, \text{ino} \rangle$ in the B^+ tree. Besides that, there is another special *finder* entry with the key $\text{path}/\backslash\text{xFE}$ for each of these components in the B^+ tree¹. The *finder* entry value denotes component type. According to B^+ tree lookup policy, if there is no equal entry for the requested key, the first entry whose key is greater than the requested key is returned. For example, if the file path is $/a/./b/\text{link}/c$ where *link* is a symlink, the associated *finder* entry key is $/a/b/\text{link}/\backslash\text{xFE}$. The pathname analyzer output is $/a/b/\text{link}/c$. Because $/a/b/\text{link}/\backslash\text{xFE}$ is the first key that is greater than $/a/b/\text{link}/c$, this *finder* entry is returned during B^+ tree lookup.

Further, there is a remaining problem in finder design: a symlink following a dot-dot component. Suppose a path is $a/\text{symlink}/..$ in which *symlink* points to b/c , and the *symlink* element is removed after pathname analyzing. In

¹The ASCII character $\backslash\text{xFF}$ is reserved for shadow entry, see §4.3

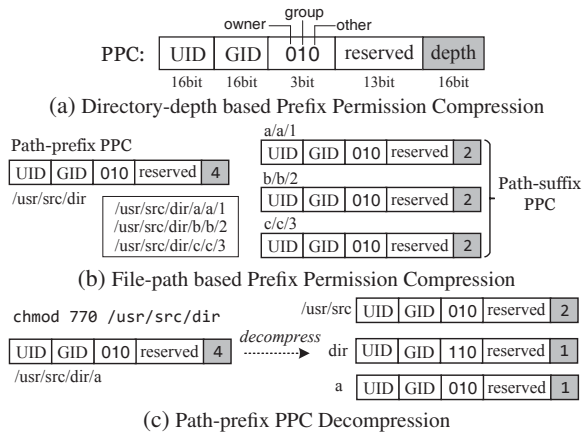


Figure 4: Two-dimensional Prefix Permission Compression

contrast, the component-at-a-time path walk will follow the symlink and generate a resolution result a/b . One solution is using the finder to check whether the current resolved path contains a symlink whenever meeting a dot-dot component during pathname analyzing. This approach degrades path walk performance if there are many dot-dot components in a file path. Another solution is using Plan 9 lexical file pathname [29]. It has better performance but causes compatibility issues. To address this dilemma, FlatFS lets users decide which file path type during file system mount.

If there are no semantic components in a path, FlatFS directly uses the pathname to search the namespace B^r tree to find the associated file inode.

Prefix permission checker. When the user accesses a file, the file system verifies whether the calling process has the execute permission for each directory listed in the file path. We call it prefix permission checking. FlatFS separates the prefix permission checking from the pathname lookup and adopts two-dimensional prefix permission compression to reduce checking performance costs.

First, directories in a file path often have same permission fields [15]: UID, GID, execute permission bit. FlatFS compresses these directory permission fields into a structure called *prefix permission compressor* (PPC for abbreviation), as shown in Figure 4a. The *depth* field denotes how many levels of directories are compressed in this PPC. Second, PPCs of those directories whose index keys are in the same B^r tree leaf node also can be compressed based on the file path (depicted in Figure 4b). The permission fields of the file path prefix are compressed into the path-prefix PPC. The permission fields of the remaining file path are compressed into the path-suffix PPC. File-path-based prefix permission compression is helpful in batching PPC updates.

A directory permission change may cause PPC decompression. Figure 4c illustrates a path-prefix PPC is decompressed into three PPCs caused by a *chmod*. Every new PPC records compressed permission fields of a sub-file path. Moreover, this decompression is propagated to PPC of all sub-directories and associated files under `/usr/src/dir` directory. Fortu-

nately, the file-path-based compression reduces update costs. If keys in a leaf node share a prefix `/usr/src/dir`, only the path-prefix PPC in tree leaf node is updated. Further, as directory permission changing are rare operations reported in a recent research paper [15], the PPC decompression incurs a slight performance impact on realistic applications.

The PPC allows batching permission checking. When performing a prefix permission checking, the task uses its credential to verify the UID and GID in both path-prefix and corresponding path-suffix PPCs. Then, it compares the directory depth of the file path and total execution bits compressed in path-prefix and path-suffix PPCs. If they are equal, file access permission is granted. For a relative file path, FlatFS extracts associated permission bits in path-prefix and path-suffix PPCs according to the path to be checked, and then performs permission checking.

The symlink and dot-dot components require careful consideration to preserve the semantic. If a symlink points to an absolute path, it may cause a namespace switch. FlatFS performs prefix permission checking on file path parts which belong to its namespace. Similarly, when adopting non-lexical file paths, FlatFS also performs prefix permission checking on path components before the dot-dot component. Overall, current POSIX prefix permission checking specification is more beneficial to the traditional path walk model.

Path dispatcher. Path components like dot-dot, symbolic link, and mount point could cause namespace switches during path walk. The path dispatcher performs a namespace switch and sends the pathname to the path walk model. Although the destination namespace may be hierarchical or flat (case I & II in Figure 3), the switch procedure is the same, mainly including mount point switch, pathname cursor adjustment, and other environment setups. In addition, If a symlink points to a relative file path (case III in Figure 3), the path dispatcher generates a new file path by resolving the symbolic link and restarts the whole path walk.

Coordination. A file path walk may involve different path walk models. These different path walk models are coordinated to resolve a pathname correctly. A path walk model can be viewed as a black box. Feeding a pathname input, it generates the resolved pathname output. The key to coordination is ensuring correct path walk model switch. Specifically, the VFS uses a pair $\langle \text{mnt point}, \text{root dir} \rangle$ to specify a mounted file system. We also create one for FlatFS instance, which is used to switch in or out its namespace. Furthermore, we add a pathname cursor for each path walk model to indicate the current resolved pathname position. This cursor feeds a correct pathname input to the path walk model.

4.2 Range-optimized Index Tree

FlatFS uses a persistent range-optimized B^r tree as its indexing data structure. The index keys of B^r tree are full pathname byte strings. Figure 7 shows the B^r tree leaf node layout. The tree node is aligned to 256 bytes, which is the optimal Optane

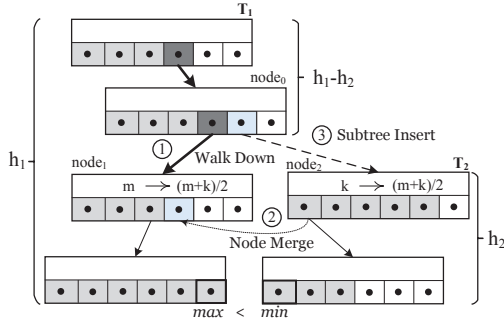


Figure 5: Tree Join Example: $T_1 \bowtie T_2 = T$

memory access granularity [46]. The keys in the B^r tree nodes are unsorted [12]. Two small pieces of metadata (a bitmap and an offset array) are added into the leaf node. B^r tree adopts a hand-over-hand locking scheme with a top-down locking order. Every tree node owns a readers-writer lock.

Directory range operations in flat namespace are more expensive than the hierarchical namespace tree [47]. B^r tree provides range operations at data structure level to overcome this challenge. The range operations are realized based on two novel tree structure primitives: *join* and *split*. We use symbol \bowtie and \ominus to denote tree join and split primitive, respectively. Also, we define tree node fanout f and total tree items N . The whole tree is locked during range operations.

Tree join. The \bowtie primitive concatenates two smaller trees T_1 and T_2 and generates a larger tree T : $T_1 \bowtie T_2 = T$. The maximum key in T_1 must be smaller than the minimum key in T_2 . Figure 5 illustrates an example of joining T_1 and T_2 . These two tree heights are h_1 and h_2 respectively, where $h_1 > h_2$. The detailed tree join steps are described as follows. First, we walk downside the T_1 from top level to the level $h_1 - h_2$ by always walking the rightmost tree node in each level (Step ①). Then, we concatenate these two trees by merging $node_{e1}$ and $node_{e2}$ (Step ②). The key numbers of these two nodes are m and k . If $m + k \leq f$, we only need to copy all keys and children from $node_{e2}$ to $node_{e1}$. Otherwise, the key number of two nodes is re-balanced as $(m + k)/2$. These associated keys and children are moved from $node_{e2}$ to $node_{e1}$. Finally, a new subtree $tree_2$ is inserted into the parent node $node_{e0}$ (Step ③). The time complexity of tree join \bowtie is $O(|h_1 - h_2|)$.

Tree split. The \ominus primitive splits the tree T into two smaller trees $LTree$ and $RTree$ for split point x : $T \ominus x = \{LTree, RTree\}$. The maximum key of $LTree$ is less than x and the minimum key value of $RTree$ is greater than or equal to x . Figure 6 depicts tree split steps with the split point 9.

At the root node level, we divide all keys in the $node_{e1}$ into two parts. The maximum key of the left part is smaller than 9. The minimum key of the right part is greater than or equal to 9. This key division also partitions the tree into two subtrees L_1 and R_1 . Then, we walk down to the $node_{e2}$ pointed by child C_1 . The key in $node_{e2}$ is in the range of $[5, 20]$. Similarly, keys in $node_{e2}$ also can be divided into two parts. As a result, two subtrees L_2 and R_2 are generated. This operation is repeated until reaching the leaf level. Finally,

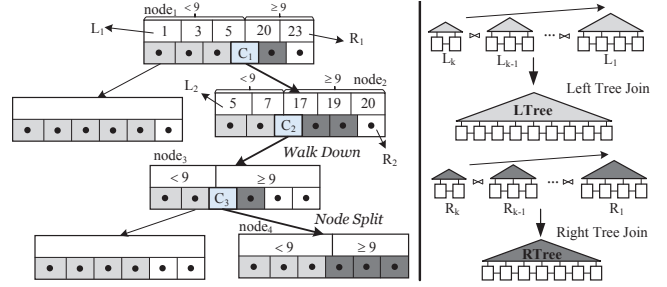


Figure 6: Tree Split Example: $T \ominus 9 = \{LTree, RTree\}$

this tree split generates two small tree sets: $\{L_1, L_2, \dots, L_k\}$ and $\{R_1, R_2, \dots, R_k\}$. Then, we perform tree join operations on all trees in each tree set in an ascending order of tree height. For example, the $LTree$ is generated by joining k subtrees: $L_1 \bowtie L_2 \dots \bowtie L_k$. Finally, the tree T is split into two trees: $LTree$ and $RTree$. The time complexity of \ominus operation is $O(\log_f N + \sum(h_i - h_j)) = O(\log_f N)$.

Range operation. B^r tree provides four range operations: *range query*, *range slice*, *range insert*, *range update*. The range query operation workflow is the same as a textbook B^+ tree. The range slice operation detaches a subtree t from the original tree T with a given key range $[k_l, k_r]$. The detailed range slice steps formulate as follows:

$$T \ominus k_l = \{p, q\} \Rightarrow q \ominus k_r = \{t, r\} \Rightarrow p \bowtie r = T' \quad (1)$$

First, we split the tree T into two parts $\{p, q\}$ for k_l using a \ominus operation. All keys in tree q are greater than k_l . Thus, the tree q is split again with the key k_r . Keys of tree t locate in the range $[k_l, k_r]$. Consequently, the tree t is the range slice result. However, the original tree T structure is destroyed by two split operations. We heal the tree T by using a \bowtie operation to concatenate the p and r . The tree range slice operation is useful in the directory remove and move operation because it can remove a bunch of indexes for targeted files and directories from the B^r tree at a time.

The tree range insert operation inserts a small tree into a large tree at a time. There are two cases for a tree range insert. To insert a tree T_2 into another tree T_1 , if all keys in T_2 are smaller than keys in T_1 , we can directly join these two trees: $T_1 \bowtie T_2 = T'_1$. Otherwise, the tree range insert operation formulates as follows:

$$T_1 \ominus key_{min} = \{T'_1, T_r\} \Rightarrow T'_1 \bowtie T_2 = T''_1 \Rightarrow T''_1 \bowtie T_r = T'''_1 \quad (2)$$

The tree T_1 is split into T'_1 and T_r with the minimum key key_{min} of tree T_2 . The keys in T'_1 , T_2 , and T_r are in ascending order: $key_{T'_1} < key_{T_2} < key_{T_r}$. This range insert operation is achieved by performing two join operations on three trees.

The tree range update modifies the index keys of a specific range. A naive solution to range update is walking the tree and updating all index keys in tree nodes. Our B^r tree uses the WoC key design to reduce the key update costs. The WoC key fetches the common part of all index keys as a key prefix. Therefore, the range update operation only modifies the key prefix without updating these keys one by one.

Leaf node caching. We design a leaf node cache to reduce tree walk performance costs. Every CPU owns a volatile in-DRAM node cache structured as a LRU list. The cache entry stores the leaf node memory address instead of the real leaf node. Hence, no data synchronization is required among multiple CPU caches. The B' tree lookup consists of a fast path and a slow path. The fast path traverses the LRU list, locks and accesses these real leaf nodes, and searches for the item with the requested key. If there is a hit, the slow path that walks the whole tree can be avoided. The leaf node cache is effective as namespace operations in the real world often exhibit good locality. For example, creating files in a directory needs to search the same directory file inode multiple times.

Every leaf node has a reference counter for safe memory reclamation. Creating a node initializes the counter as one, and the cache insertion increases the counter. Because the cache is volatile, no crash safety is needed for counters. They will be reset as one during remount. A cache entry may refer to a removed tree node. The removed node with a non-zero counter is kept in a persistent list for lazy reclamation. Accessing the removed node is safe since it contains no valid keys due to the empty bitmap. This node will be recycled when the associated cache entry is evicted.

4.3 Directory Range System Call

Besides *rename*, FlatFS designs three new system calls for directory range operations.

Directory read. FlatFS offers a *getdents_recur* system call. This system call lists all files and subdirectories in a directory recursively (similar to *ls -R*). To support recursive range query for a directory, FlatFS introduces two shadow entries for each directory. Their filenames are the first and last ASCII character. Keys of all entries in the directory are delimited by these two shadow entry keys. To perform a recursive directory reading, FlatFS uses the small shadow entry key to lookup the tree to find the leaf node. Then, it repeats this with the other large shadow entry key and fetches a range of keys at a time. For non-recursive directory reading, FlatFS uses a directory skip approach. When FlatFS meets the first shadow entry of a subdirectory during directory scanning, it skips entries in the subdirectory by performing a tree lookup with the other shadow entry key.

Directory remove. FlatFS introduces a new system call *rmdir_recur*, which is used to remove all files and subdirectories in a directory at a time. In the data structure layer, we use a range slice to obtain the subtree that contains entries to be removed from the B' tree. Then, we perform a range query to this subtree, find inodes of these files and directories, remove files and directories at the file system layer. To further reduce the *rmdir_recur* latency, we also delay freeing the sliced subtree structure.

Directory copy. FlatFS provides a new system call *cpdir_recur(src, dst)*, which copies a directory recursively to another directory. First, the subtree of *src* directory is sliced

and duplicated. The index key updates of the duplicated tree are batched. Then, both sliced and duplicated subtrees are inserted into the B' tree. Then, new metadata are created for copied files and directories. We follow the default semantic of *cp -r* for hard links, i.e., the linking becomes invalid. A new inode is allocated for the copied hard link and its file content is the same as the original linked file. Besides, FlatFS also corrects the inode number of dot and dot-dot of every directory, making them point to the newly created directory and its parent directory. Another performance optimization adopted by *cpdir_recur* is file data copy-on-write mechanism. We only duplicate the file mapping of every copied file. The last level pointers in the file mapping are set as copy-on-write.

Directory move. The directory rename mainly involves data structure level operation. Similar to directory remove, the subtree of the *src* directory is sliced. Then, all index keys of the subtree are updated with the *dst* directory. Depicted in Figure 7, every tree node contains an index key prefix. A key update traverses the B' tree and iterates over every node. If the *src* directory pathname is a substring of the key prefix in this node, this key update just modifies the key prefix with the *dst* directory pathname. Finally, the updated subtree is inserted into the B' tree.

4.4 Write-optimized Compressed Key

FlatFS uses key compression to reduce storage consumption and key batch update costs. Every variable-size index key is divided into a prefix and a suffix. All index keys in the same tree node share a prefix. However, key compression incurs performance overheads due to prefix and suffix adjustments. For example, inserting a key would shrink the prefix and expand all suffixes, leading to many small data writes, which causes expensive cache line flushes and write amplification.

Basic idea. We propose WoC key to address this challenge. WoC key prefetches a number of characters during suffix expansion. These data are cached in the suffix to avoid future data movements. Moreover, we also record the prefix size and the total size of each key. Suppose another key insert event causes all suffixes to expand and suffix size increasing. Fortunately, we only need to update the prefix size to represent all suffix data and size changes.

Furthermore, removing a key may cause all suffixes to shrink and the prefix to expand. There are no data movements for suffix shrinking. We append data to the prefix and increase the prefix size. There is no data prefetch optimization for the prefix because a tree node only has one prefix, and its adjustment cost is low. In contrast, every node entry owns a suffix. Suffix changes lead to high memory write costs.

Detailed Design. Figure 7 shows the key suffix is partitioned into inuse area and cache area. The *key suffix addr* stores suffix memory addresses. The inode number is stored along with the suffix. The *woc-length array* contains a set of eight-bytes *woc-length* structures. Every index key owns a suffix and a *woc-length* structure. The total key size *key_total_sz*

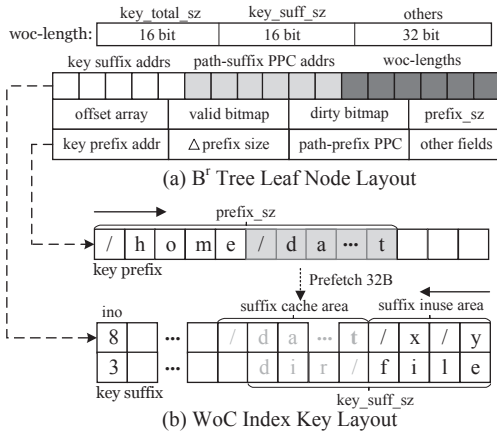


Figure 7: B^+ Tree Leaf Node and Index Key Layout

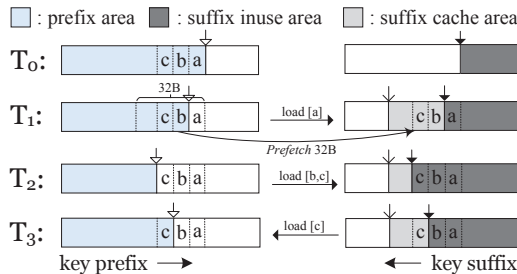


Figure 8: Illustrative Examples of Suffix Prefetch and Caching and the suffix size key_suff_sz are stored in the woc_length . The key_suff_sz is the sum of suffix inuse and cache area size. The inuse area size is $key_total_sz - prefix_sz$. If the cache area size is smaller than the data to be appended, it indicates the suffix cache is insufficient and requires another prefetch.

Figure 8 illustrates three examples of how to reduce data movements with WoC key design. At time T_1 , one-byte data a is moved to the suffix area. Instead of moving one character, we prefetch 32-bytes of data from the prefix memory area to the suffix memory area. Then we modify $prefix_sz$ and key_suff_sz and persist these two fields. 32 bytes is the optimal data prefetch granularity according to evaluation results. Prefetching 64-bytes data may introduce two cache line movements if the first byte of prefetched data is not cache-line aligned. On the other side, a smaller prefetch granularity introduces more frequent data movements.

The prefetched data are stored in the cache area to avoid future data movements. For instance, at time T_2 , two bytes of data $[b, c]$ are moved to the suffix area. Fortunately, these two-bytes data are already stored in the suffix cache area. We only need to update the $prefix_sz$ field to indicate this key suffix expansion. When the key suffix shrinks at time T_3 , the associated data c is appended to the key prefix. There are also no data movements for this suffix shrinking event.

The key batch update introduces two issues to WoC key design: (1) if a key batch update modifies the key prefix, the key_total_sz fields of all key suffixes should be updated; (2) a key batch update may cause cached data in the suffix to become stale due to key prefix change. To solve the first problem, we introduce a $\Delta prefix_size$ field, which denotes

the key size changes to all index keys in a tree node caused by a batch update. Instead of updating all key_total_sz fields, the key batch update modifies the $\Delta prefix_size$. The key suffix inuse area size is $key_total_sz - prefix_sz + \Delta prefix_size$.

The second problem is addressed by introducing a dirty suffix bitmap. The bitmap records those key suffixes whose cached data become stale in the index key batch update. We query the dirty bitmap before adjust the key suffix. If the associated suffix cache area is dirty, we clean the suffix cache area by resetting key_suff_sz as the suffix inuse area size.

4.5 Namespace Crash Consistency

Existing metadata system calls except for *rename* only introduce tree point operation. The general workflow of these system calls consists of three steps: (1) pathname lookup; (2) inode manipulation; (3) tree point operation. Achieving crash consistency for these system calls is similar. We take *creat* system call as an example to describe the approach.

First, the newly allocated inode is logged at the file system layer. Then, FlatFS inserts a new entry into B^+ tree. This entry insert may be a simple leaf node insert or requires a node split. In the worst case, a leaf node insert causes a key prefix adjustment, a new key suffix write, a cascade of updates to other key suffixes, and a valid bitmap update. Thanks to the WoC key design, we only needs to log 18-bytes data (2-bytes prefix size, 8-bytes prefix address, and 8-bytes valid bitmap). Logging key prefix size and address ensures data consistency for both key prefix and suffix. To be specific, all old suffix sizes can be restored with logged old prefix size. Then we reset key_suff_sz as key suffix inuse area size. The key prefix is also restored using logged old prefix address. In addition, if there is a node split for this entry insert, the valid bitmaps of the split node and parent node are also logged.

Ensuring metadata consistency is challenging for directory range operations as they introduce complicated tree structure manipulation. We first explain how to ensure crash-safety for B^+ tree range operations. Depicted in Figure 5, a tree join involves three nodes: two merged nodes and a parent node. The valid bitmaps of these nodes are logged. Ensuring crash consistency for tree splits is more difficult. Supposing the tree T height is h , the \ominus operation splits h nodes at each tree layer with split point x . The valid bitmap of h nodes is logged. Next, two new trees L and R are generated by joining these split trees in two sets separately. It seems difficult to recover the tree state if a crash occurs during tree joins. A useful property between \bowtie and \ominus simplifies the crash-consistency implementation, i.e., \ominus is the reverse operation of \bowtie . The original tree T can be recovered by joining L and R : $L \bowtie R = T$. Thus, root node memory addresses of joined trees (i.e., $\{L_1, \dots, L_k\}$ and $\{R_1, \dots, R_k\}$) are logged. During recovery, we replay tree join operations with logged root node addresses and re-generate the namespace tree T with L and R .

The tree *range insert* and *range slice* are implemented based on tree join and split. We already guarantee crash-

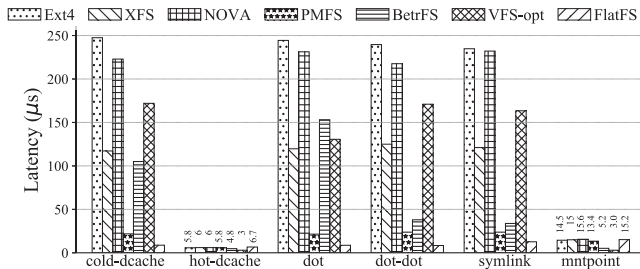


Figure 9: Path Walk Efficiency

safety for tree \bowtie and \ominus . Crash consistency of these two tree operations is achieved based on an important insight. Even the original namespace tree is temporarily decomposed into many pieces due to split/join, however, it still can be rebuilt by joining these subtrees during recovery. Finally, FlatFS also logs these affected inodes for directory range operations.

5 Evaluation

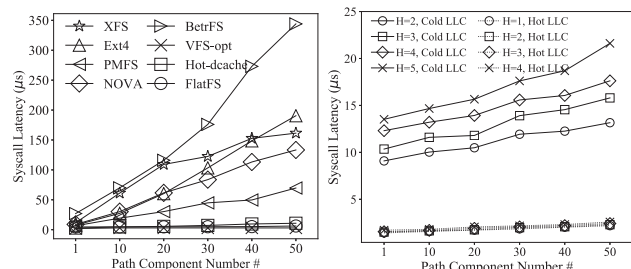
Our testbed machine has two Intel Xeon Gold 5220R processors. Each processor has 24 physical cores running at 2.2 GHz. Every physical core has a private 32 KB L1 cache and a 1 MB L2 cache. All cores that resided on the same socket also share a 35.75 MiB last level cache. The hyper-threading is disabled. Each processor has two integrated memory controllers. The machine has 192 GB (12×16 GB) DRAM, 1.5 TB (12×128 GB) Intel Optane DC persistent memory, and a 512 GB Solid State Driver.

We implement FlatFS in Linux kernel 4.15 based on PMFS file system [31]. FlatFS reuses PMFS data path and journal mechanism. We compare FlatFS with four NVM file systems (NOVA [44], PMFS [31], Ext4 [5] and XFS [37]), a full-path-indexing file system BetrFS [47], a VFS dcache optimized system (VFS-opt) [40]. BetrFS and VFS-opt work on Linux 3.11.10 and 3.14, respectively. These old kernels do not support Intel Optane memory. We use Linux *brd* module to create a RAM disk for BetrFS and VFS-opt. The RAM disk uses the fast DRAM to emulate block devices. The device performance is better than the Optane memory. We also mount an Ext4 file system on the RAM disk for VFS-opt.

5.1 Microbenchmark Performance

Path walk efficiency. We evaluate the path walk performance with different kinds of path components. We *stat* a file whose path comprises nine components in the Linux kernel 4.4 source code directory. When dcache is cold, file systems fetch data from the device, which causes long latency. This problem is especially serious for BetrFS. BetrFS adopts a stacked system architecture. A dentry missing event is handled by multiple software layers. Its path walk latency is $11.93 \times$ higher than FlatFS. The sequential file access results in a hot dcache, which greatly reduces path walk latency. Fortunately, FlatFS also achieves low latency thanks to its node cache design. The VFS-opt delivers the lowest latency ($3 \mu s$) for its hash-based full-path-indexing optimization.

VFS and FlatFS handle dot components lexically. Thus, the



(a) Path Walk Scalability

(b) Path Walk Sensitivity

Figure 10: Path Walk Scalability and Sensitivity

Table 2: Path Sensitivity to Sequential and Random Access. H: B^r Tree Height; L: File Path Length; N: File Number

Setting	H=3, N=10 ⁴ , L=20		H=4, N=10 ⁵ , L=20		H=5, N=10 ⁶ , L=20	
System	VFS	FlatFS	VFS	FlatFS	VFS	FlatFS
Seq.	2.05 μs	1.63 μs	2.04 μs	1.63 μs	2.05 μs	1.63 μs
Rnd.	2.11 μs	2.19 μs	2.24 μs	2.54 μs	2.26 μs	3.66 μs

path walk latencies are similar to the cold dcache case. FlatFS performs semantic path component checking for dot-dot and symlink components, which incurs an extra namespace tree query. Even so, FlatFS performs 1.87 - $28.52 \times$ better than others for its non-caching namespace design. Finally, we mount an Ext4 file system under each tested file system. The mount operation causes a hot dcache for path walk. The performance results are similar to the hot dcache case. Overall, for a nine-component path, FlatFS outperforms other file systems significantly for the cold dcache and delivers similar latencies for the hot dcache.

Path walk scalability. We vary path component number from 1 to 50 to evaluate path walk scalability using *stat* syscall. As shown in Figure 10a, when the file path becomes lengthy, the execution latencies of five file systems (i.e., BetrFS, XFS, Ext4, PMFS, NOVA) also increase significantly. The FlatFS path walk latency ($5.1 \mu s$) is stable against different path lengths. The VFS-opt system achieves a low constant path walk latency ($3 \mu s$). We also measure the path walk scalability of hot dcache. When the path component is less than 10, its latency is close to ours. However, its latency increases $5 \times$ ($10.8 \mu s$) varying from 1- to 50- component.

Path walk sensitivity. We evaluate FlatFS path walk sensitivity by changing four variables: B^r tree height, Last-Level-Cache (LLC), path component number, and access pattern. We create four file sets of different sizes (10^3 , 10^4 , 10^5 , 10^6) with tree height H ranging from 2 to 5. We vary the path length and measure the syscall latency. We make three observations from Figure 10b. First, hot LLC significantly boosts FlatFS path walk performance. Second, path walk performance is sensitive to tree height, especially for cold LLC. Resolving a 50-component path in a 5-level B^r tree takes $21.61 \mu s$, which is $1.64 \times$ longer than 1-level B^r tree. Third, path length has a slight impact on FlatFS path walk performance. A 50-component path walk is $1.5 \times$ longer than 1-component. In contrast, other file systems deliver $9.8 \times$ - $16.3 \times$ latency increment in Figure 10a.

Table 3: Directory Range Operation Latency (s)

	Ext4	XFS	NOVA	PMFS	BetrFS	VFS-opt	FlatFS
readdir	0.152	0.349	0.208	0.097	0.388	0.157	0.031
rmdir	0.61	1.262	1.131	0.548	3.680	0.736	0.190
cpdir	2.398	2.907	2.334	1.949	4.652	1.829	0.450
mmdir	0.004	0.004	0.004	0.004	0.019	0.004	0.007

Table 2 shows path walk latency of sequential and random file access. The LLC and VFS dcache is warmed before experiments. FlatFS outperforms VFS for sequential access in three settings. The reason is batched operations in FlatFS path walk. To confirm it, we use the perf tool to collect instruction numbers during syscall execution in setting S_3 . The profiling results report that VFS executes 1.43× more instructions than FlatFS. Random access has a larger impact on FlatFS than VFS. VFS gains much more benefits from its cached dentries of the prefix path. Random access causes 3.24× more cache misses for FlatFS than VFS. As the file size decreases, FlatFS also achieves better performance for higher LLC hit ratio. Finally, sequential access outperforms random access due to higher node cache hit ratio.

Directory range operation. Table 3 shows four common directory range operation performance. FlatFS uses four directory range system calls. For other file systems, we use *ls -R*, *rm -r*, *cp -r*, and *rename* instead. All directory range operations manipulate a Linux kernel 4.4 source code tree.

Reading a directory recursively is well-optimized in FlatFS. FlatFS reduces directory read latency by up to 12.52× compared to others. These file systems except BetrFS scatter the dentries in data blocks across the storage device, which greatly hurts the data locality. For directory removing, FlatFS obtains performance gains from 2.88× to 19.37× relative to other file systems. Since namespace traversal is fast in FlatFS, it benefits directory remove as FlatFS only needs to perform a simple scan to obtain the entries for de-allocation. Other file systems require expensive tree traversal to retrieve target inodes. Also, FlatFS delays freeing the directory subtree to further reduce the latency.

FlatFS outperforms other file systems by 4.06×-10.34× for directory copy. Because the directory copy is data I/O dominated. The file data copy-on-write optimization in FlatFS effectively avoid such performance costs. BetrFS is well-optimized for HDDs. For ultra-fast NVMs, its stacked architecture is suboptimal. We find interactions between BetrFS and the underlying storage device involve many software layers, which causes high software latencies due to both additional and duplicated works. Directory move is the only range operation provided by conventional file systems. Directory move is cheap for the hierarchical namespace structure, which only takes 4μs for moving a Linux directory tree. FlatFS also achieves a low rename latency 7μs using tree-level range operation and batching key update.

5.2 Macrobenchmark Performance

Filebench. We choose two application-level workloads: fileserver and varmail from Filebench [38]. We create a file set

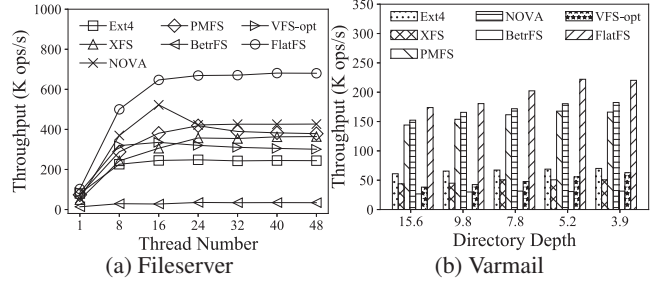


Figure 11: Performance of Filebench Benchmark

with 100 thousand 4KB files for the fileserver. We stress the namespace performance by setting the average directory width as two. Figure 11a shows the fileserver throughput varying thread number from 1 to 48. FlatFS achieves a 1.79 performance speedup over PMFS. FlatFS and PMFS share the same file data path but differ in metadata management. When thread number increases, PMFS suffers from poor scalability due to its centralized inode table design. NOVA and FlatFS address this issue with per-CPU inode table design. NOVA performs worse than FlatFS for its low namespace insert/remove performance and fault tolerance costs [45]. NOVA uses a Radix tree to manage entries in the directory. A *create/unlink* inserts/removes the associated entry in the parent directory. Because a directory only has two files, file creation/removal incurs high tree management costs.

The centralized journal in Ext4 also causes severe scalability issues [19,35]. VFS-opt requires Ext4 to create files, which accounts for its low scalability. VFS-opt achieves 23% higher throughput than Ext4 for its dcache optimization. Finally, BetrFS exhibits the lowest performance. Since the cache is hot during execution, we explain it as NVM-unaware B^e tree design. B^e tree adopts big internal (4MB) and leaf nodes (4-11 MB). Moreover, it uses an order maintenance tree (OMT) to organize entries inside the leaf node. The entry insert or remove is expensive because it introduces many entry movements and OMT adjustment costs during file create and remove. In contrast, FlatFS reduces key management costs during tree insert/remove with its write-optimized key design.

In varmail experiment, we create a file set that contains 50 thousand 64-byte files to simulate a large set of small e-mails in the mailbox. We vary the *meandirwidth* parameter from 2 to 16. It affects the directory depth of the tested file set. When the directory depth increases, the path walk time increases. Therefore, the throughput of all file systems decreases. FlatFS obtains 14.10%-3.97× higher throughputs than other file systems due to its fast path walk design. Moreover, varmail workload frequently creates and deletes files. Besides fast path walk, FlatFS also gains performance benefits from its high namespace insert/remove performance.

FxMark. We pick MWCM and MWUM benchmarks from FxMark [25] to measure the file system multicore scalability. Figure 12a and Figure 12b shows experimental results. We draw three conclusions from these experiments. First, FlatFS performs much better than other file systems in file creation

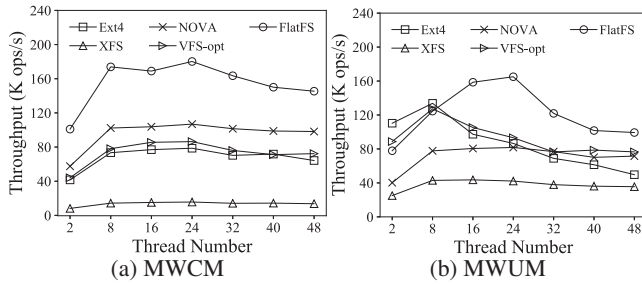


Figure 12: Scalability of FxMark Benchmark

and removal, which achieves a maximum $2\times$ performance speedup. Other file systems exhibit low namespace insert/remove performance due to inefficient dentry index structure design. For instance, PMFS uses a linear array to organize entries in a directory [31]. When there are a large number of files, searching an entry in the linear array takes long time. We even fail to run these two benchmarks on PMFS and BetrFS due to their extremely low dentry search performance.

Second, the inode lock is the major scalability bottleneck for file creation and removal. All in-kernel file systems including FlatFS adopt the same inode locking scheme. Thus, their performance trends with different thread numbers are similar. However, FlatFS still outperforms others for its higher namespace performance. Third, NUMA impacts greatly affect file system scalability. Both Figure 12a and Figure 12b show that all file systems experience a degradation when there exists remote memory node access beyond 24-threads.

5.3 Factor Analysis of Each Optimization

We analyze the performance benefits of three optimization techniques in FlatFS.

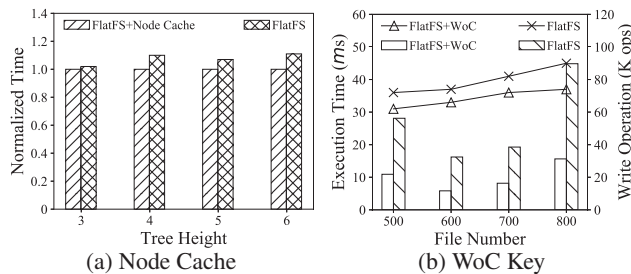


Figure 13: Performance Benefit of Node Cache and WoC Key

Node caching optimization. The node cache design avoids tree traversal for entries in the same tree node. It is especially useful for sequential file access. We create four large directories containing 10^4 , 10^5 , 10^6 , 10^7 files, respectively. The corresponding B^+ tree height ranges from three to six. We create a microbenchmark to access all files in the directory. Figure 13a shows FlatFS achieves 2.01%-11.07% performance improvements with node caching optimization. In addition, when tree height increases, FlatFS obtains more benefits from tree node caching.

Rename optimization. The rename operation is cheap and fast in directory tree-based namespace. It takes 0.004s to move a Linux-4.4 source code repository. We measure two

rename implementations in FlatFS. The slow *rename* implementation removes all associated entries in the flat namespace, updates pathname keys, and then inserts all entries. Its latency is ten times slower than conventional file systems. Fortunately, FlatFS improves *rename* performance with tree range operation. Its optimized *rename* takes 0.007s.

WoC key design. We use a microbenchmark to measure the performance benefits of WoC key design. The microbenchmark creates a large number of files in a directory. File creation causes index key prefix and suffix adjustments, which incurs high data persistence costs. Figure 13b demonstrates that FlatFS+WoC delivers average 15.94% lower latency than FlatFS without WoC key optimization. We also use IPMWatch tool [9] to collect the number of writes received by the memory controller. Figure 13b shows WoC key design achieves a nearly $2.64\times$ write reduction.

5.4 Version Control System: Git

We demonstrate FlatFS performance benefits with git application. We create a large deep directory tree containing ten Apache Hadoop 2.10.1 source code repositories [3]. The average directory depth of all files is 17.71. We use two frequently used git commands `git status` and `git commit` to evaluate file system directory reading and path walk performance. In the original `git status` implementation, it traverses the directory tree of the target repository recursively, reads every directory entry, validates its state, and puts it into the associated list according to its state (e.g., *untracked*). Figure 14a shows that FlatFS outperforms others by up to $2.21\times$.

Besides, we also modify the `git status` implementation using the `getdents_recur` syscall in Section 4.3 (denoted as `Git-opt+FlatFS` in Figure 14a). The `getdents_recur` system call performs a range query to the target directory and sub-directories and returns all entries at a time. It greatly improves `git status` performance by eliminating expensive hierarchy tree traversal. The optimized `git status` only takes 0.6s, which reduces the latency by $4.12\times$ compared to the unmodified `git status` command.

The `git commit` first uses `lstat` to check the file existence, then it opens every file and reads its content using a `mmap` system call. The massive number of `lstat` and `open` syscall in `git commit` greatly stresses the path walk efficiency. The experimental results show that FlatFS reduces the `git commit` latency by 13.79%-37.50% compared to others.

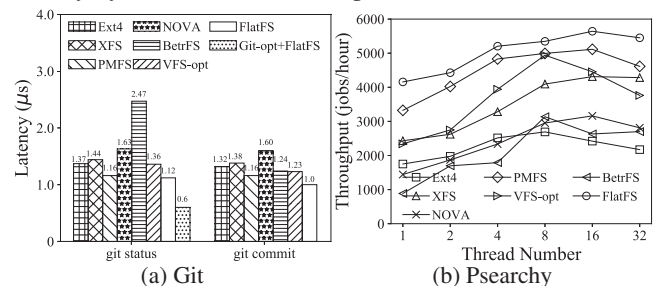


Figure 14: Git and Psearchy Evaluation Results

5.5 File Indexer: Psearchy

Psearchy is a parallel version of searchy [11]. The Psearchy creates multiple worker threads. Each thread processes a number of files. Specifically, the thread opens every file, reads file content, records word positions in a local hash table, sorts word positions, and persists results in a Berkeley DB file. We set the local hash table size 16 MB. We also create a dataset using the taxonomy of known animals from the Catalogue of Life website [4]. The dataset directory hierarchy is generated according to the hierarchy of biological classification. It contains 276,616 files and directories. Because the file size is small, the psearchy performance is largely dependent on file open performance. Figure 14b shows that applications throughput increases as the thread number increases. Among them, FlatFS always achieves the highest throughputs. However, scalability issues caused by the glibc *qsort* function and VFS inode hash table lock prevent throughput improvements as thread number increases.

Table 4: Total and *stat* Syscall Execution Time of Hive

	Ext4	XFS	NOVA	PMFS	BetrFS	VFS-opt	FlatFS
Total	8.44s	8.71s	8.55s	8.50s	8.39s	8.34s	8.28s
<i>Stat</i>	1.46s	1.40s	1.52s	1.34s	-	-	1.18s
μ s/Call	3.07	2.94	3.20	2.81	-	-	2.50
Gain	18.4%	15.1%	21.8%	11.0%	-	-	

5.6 Data Warehousing System: Hive

Apache Hive is a data warehousing system that uses database table partition to improve data query performance [39]. We use TPC-H benchmark [8] to create a database table for evaluation. Then we use table partition technique to generate a dataset containing 74,090 files and directories. We create a Hive SQL benchmark to perform queries to all files. The major performance bottleneck of Hive is the Java virtual machine. Therefore, Table 4 shows FlatFS slightly outperforms other file systems in total execution time.

During Hive execution, We find that *stat* is the most frequently invoked metadata syscall. To measure the performance benefits of our namespace design, we uses the *strace* tool [7] to collect the *stat* syscall execution time. We fail to profile BetrFS and VFS-opt due to unknown bugs. Table 4 reports that FlatFS achieves 16.58% better performance than others in *stat* syscall for its optimized file path walk.

6 Related Works

File Metadata Indexing Optimization. TableFS [32] and BetrFS [18] share similar idea that utilize existing key-value stores (LevelDB and TokuDB) to improve metadata indexing performance. TableFS [32] aggregates file metadata in a LevelDB table indexed by a combination of parent directory inode number and filename string. BetrFS utilizes the TokuDB to optimize both metadata and data block indexing performance with their full-path indexing schema [18]. These two file systems achieve good performance running on hard disks. However, they introduce new software layers into the

existing storage stack. Our experimental results show that such stacked system architecture is inefficient for file systems built for fast NVMs. FlatFS exhibits a non-caching system architecture to avoid data copy costs for ultra-fast NVMs.

File Path Walk Optimization. ByVFS [41] bypasses the VFS layer during path component resolution and directly fetches dentries from the device. Tsai et al. [40] propose two techniques to improve file path walk performance in the VFS layer. First, they reduce the pathname lookup latency using full-path indexing via an in-memory hash table. Second, they reduce the dcache hit latency by caching the permission results of file paths. DLFS [21] re-organizes the on-disk metadata and proposes a hashing-based metadata indexing solution to improve pathname lookup performance. Directory range operations (e.g., *rename*) are expensive in their system. FlatFS incorporates a full-path-at-a-time path walk model to accelerate pathname lookup performance.

Namespace Structure Optimization. ReconFS [22] decouples the file system namespace as a volatile and a persistent directory tree. It improves system performance and device endurance by reducing metadata writes to flash memory. Partition is an efficient solution to improve namespace scalability. IndexFS [33] proposes a scalable directory service based on GIGA+ [27] to dynamically partition a large directory tree across multiple nodes in the distributed environment. SpanFS [19] is a scalable local file system that distributes the global namespace into multiple disjoint domains to avoid contention caused by centralized namespace management. FlatFS exploits the flat namespace structure to optimize the namespace lookup and reading operation performance.

7 Conclusion

This paper demonstrates a novel NVM file system FlatFS. FlatFS exploits flat namespace architecture to improve metadata operation performance by proposing three techniques: a coordinated path walk, a range-optimized B^+ tree, and a write-optimized index key layout. Extensive experiments demonstrate the performance benefits of FlatFS to metadata-intensive benchmarks and applications. FlatFS is open source at <https://github.com/miaogecm/FlatFS.git>.

Acknowledgments

We thank the reviewers for their helpful feedback. We especially thank our shepherd Haris Volos for the careful, thorough reading of our paper and valuable suggestions to improve this paper substantially. This paper is supported by Fundamental Research Funds for the Central Universities (No. B220202073, B210201053), National Natural Science Foundation of China (No. 61832005, 61872171), CCF-Huawei Innovation Research Plan (No. CCF2021-admin-270-202101), Natural Science Foundation of Jiangsu Province (No. BK20190058), Future Network Scientific Research Fund Project (No. FNSRFP-2021-ZD-7), Jiangsu Planned Projects for Postdoctoral Research Funds (No. 2021K635C). Baoliu Ye is the corresponding author.

References

- [1] Intel 3D XPoint. <https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/>, 2015.
- [2] Google LevelDB. <https://github.com/google/leveldb>, 2018.
- [3] Apache Hadoop Downloads. <https://hadoop.apache.org/releases.html>, 2021.
- [4] Catalogue of Life. <https://www.catalogueoflife.org/data/download>, 2021.
- [5] Ext4 Disk Layout. <https://ext4.wiki.kernel.org/index.php/>, 2021.
- [6] Percona TokuDB. <https://www.percona.com/software/mysql-database/percona-tokudb>, 2021.
- [7] Strace: Trace System Calls and Signals. <https://man7.org/linux/man-pages/man1/strace.1.html>, 2021.
- [8] TPC-H. <http://www.tpc.org/tpch/>, 2021.
- [9] Intel Persistent Memory Watch. <https://github.com/intel/intel-pmwatch>, 2022.
- [10] Daniel P. Bovet and Marco Cesati. *Understand the Linux Kernel*. O'Reilly Media, 2006.
- [11] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Tappan Morris, and Nikolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, 2010.
- [12] Shimin Chen and Qin Jin. Persistent B+-Trees in Non-Volatile Main Memory. *Proceeding of VLDB Endowment*, 8(7):786–797, 2015.
- [13] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *USENIX Conference on File and Storage Technologies*, pages 81–95, 2021.
- [14] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. Better I/O through Byte-addressable, Persistent Memory. In *ACM Symposium on Operating Systems Principles*, pages 133–146, 2009.
- [15] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-space NVM File System. In *ACM Symposium on Operating Systems Principles*, pages 478–493, 2019.
- [16] Mingkai Dong and Haibo Chen. Soft Updates Made Simple and Fast on Non-volatile Memory. In *USENIX Annual Technical Conference*, pages 719–731, 2017.
- [17] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the Idiosyncrasies of Real Persistent Memory. *Proceeding of VLDB Endowment*, 14(4):626–639, 2020.
- [18] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuzmaul, and Donald E. Porter. BetrFS: A Right-Optimized Write-Optimized File System. In *USENIX Conference on File and Storage Technologies*, pages 301–315, 2015.
- [19] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A Scalable File System on Fast Storage Devices. In *USENIX Annual Technical Conference*, pages 249–261, 2015.
- [20] Takayuki Kawahara. Scalable Spin-Transfer Torque RAM Technology for Normally-Off Computing. *IEEE Design & Test of Computers*, 28(1):52–63, 2011.
- [21] Paul Hermann Lensing, Toni Cortes, and André Brinkmann. Direct Lookup and Hash-based Metadata Placement for Local File Systems. In *International Systems and Storage Conference*, pages 1–11, 2013.
- [22] Youyou Lu, Jiwu Shu, and Wei Wang. ReconFS: a Reconstructable File System on Flash Storage. In *USENIX conference on File and Storage Technologies*, pages 75–88, 2014.
- [23] Jim Mauro and Richard McDougall. *Solaris Internals: Core Kernel Components*, volume 1. Prentice Hall Professional, 2001.
- [24] Marshall Kirk McKusick, George V Neville-Neil, and Robert NM Watson. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2015.
- [25] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding Manycore Scalability of File Systems. In *USENIX Annual Technical Conference*, pages 71–85, 2016.
- [26] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A High Performance File System for Non-Volatile Main Memory. In

- European Conference on Computer Systems*, pages 1–16, 2016.
- [27] Swapnil Patil and Garth A. Gibson. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *USENIX Conference on File and Storage Technologies*, pages 177–190, 2011.
- [28] Ivy Bo Peng, Maya B. Gokhale, and Eric W. Green. System Evaluation of the Intel Optane Byte-addressable NVM. In *International Symposium on Memory Systems*, pages 304–315, 2019.
- [29] Rob Pike. Lexical File Names in Plan 9, or, Getting Dot-Dot Right. In *USENIX Annual Technical Conference*, pages 85–92, 2000.
- [30] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable High Performance Main Memory System using Phase-Change Memory Technology. In *International Symposium on Computer Architecture*, pages 24–33, 2009.
- [31] Dulloor Subramanya Rao, Sanjay Kumar, Anil S. Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *European Conference on Computer Systems*, pages 1–15, 2014.
- [32] Kai Ren and Garth A. Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *USENIX Annual Technical Conference*, pages 145–156, 2013.
- [33] Kai Ren, Qing Zheng, Swapnil Patil, and Garth A. Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248, 2014.
- [34] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage*, 9(3):1–32, 2013.
- [35] Yongseok Son, Sunggon Kim, Heon Y. Yeom, and Hyuck Han. High-Performance Transaction Processing in Journaling File Systems. In *USENIX Conference on File and Storage Technologies*, pages 227–240, 2018.
- [36] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The Missing Memristor Found. *Nature*, 453(7191):80, 2008.
- [37] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *USENIX Annual Technical Conference*, pages 1–14, 1996.
- [38] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *Usenix Magazine*, 41(1), 2016.
- [39] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *Proceeding of VLDB Endowment*, 2(2):1626–1629, 2009.
- [40] Chia-che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter. How to Get More Value from Your File System Directory Cache. In *ACM Symposium on Operating Systems Principles*, pages 441–456, 2015.
- [41] Ying Wang, Dejun Jiang, and Jin Xiong. Caching or Not: Rethinking Virtual File System for Non-Volatile Main Memory. In *USENIX Workshop on Hot Topics in Storage and File Systems*, 2018.
- [42] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and Modeling Non-Volatile Memory Systems. In *International Symposium on Microarchitecture*, pages 496–508, 2020.
- [43] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon D. Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. An Early Evaluation of Intel’s Optane DC Persistent Memory Module and its Impact on High-Performance Scientific Applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–19, 2019.
- [44] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *USENIX Conference on File and Storage Technologies*, pages 323–338, 2016.
- [45] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Silva, Steven Swanson, and Andy Rudoff. NOVA-fortis: a Fault-tolerant Non-Volatile Main Memory File System. In *ACM Symposium on Operating Systems Principles*, pages 478–496, 2017.
- [46] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *USENIX Conference on File and Storage Technologies*, pages 169–182, 2020.
- [47] Yang Zhan, Alexander Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and

Jun Yuan. The Full Path to Full-Path Indexing. In *USENIX Conference on File and Storage Technologies*, pages 123–138, 2018.

- [48] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: a Tiered File System for Non-Volatile Main Memories and Disks. In *USENIX Conference on File and Storage Technologies*, pages 207–219, 2019.

Artifact Appendix

A.1 Abstract

We provide FlatFS artifact description in this section. FlatFS is a metadata-optimized NVM file system that features a flat namespace. This section describes (1) how to build FlatFS on NVM systems §A.2; (2) how to reproduce the main experimental results of our paper §A.3.

A.2 How to Build FlatFS

This section describes software requirements for FlatFS and how to build FlatFS.

- **OS version:** Ubuntu 18.04 or Ubuntu 14.04.6
- **Kernel version:** Linux 4.15.0

1. Download FlatFS

```
$ git clone https://github.com/miaogecm/FlatFS.git
```

This repository contains FlatFS source code which locates in `linux-4.15/fs/flatfs` and a modified virtual file system that supports coordinated path walk.

2. Compile and Install FlatFS

```
$ make localmodconfig  
$ make menuconfig
```

Modify the Linux kernel configuration file, and make sure these two configurations are enabled:

```
File systems/DAX support  
File systems/FlatFS
```

and disable these four configurations:

```
Security options/AppArmor support  
Security options/Yama support  
Security options/TOMOYO Linux support  
Security options/Security hooks for pathname based  
access control
```

Compile the kernel

```
$ make -j
```

Install the new kernel:

```
$ make install  
$ make modules_install  
$ update-grub
```

Reboot the system:

```
$ reboot
```

3. Mount FlatFS

FlatFS can be mounted on a real or an emulated NVM device. Create a mount directory and mount FlatFS:

```
$ mkdir /mnt/flatfs  
$ mount -t flatfs -o init /dev/pmem0 /mnt/flatfs
```

4. Umount FlatFS

```
$ umount /mnt/flatfs
```

A.3 Experiment Reproducibility

We provide a number of helpful scripts to reproduce the main experimental results automatically. Specifically, readers could reproduce Figure 9, Figure 10a, Figure 10b, Figure 11a, Figure 11b, Figure 14a, Figure 14b, Table 2, Table 3, and Table 4.

Reproducing an experiment takes three steps except for the Hive experiment.

Step 1. Clean old data:

```
$ ./clean
```

If you are reproducing Table 4, generate data for the experiment first:

```
$ export TBL_PATH=~/.hive/table  
$ ./mktable
```

Step 2. Collect data for each tested file system, where FS=ext4, xfs, pmfs, nova, flatfs, betrfs, vfs_opt. The generated data is saved in a `.data` file in the current directory.

```
$ ./run $FS
```

Step 3. Draw the figure with collected data:

```
$ ./plot.py
```

More details can be found in `evaluation/README.md` in the repository.

StRAID: Stripe-threaded Architecture for Parity-based RAIDs with Ultra-fast SSDs

Shucheng Wang¹, Qiang Cao^{1*}, Ziyi Lu¹, Hong Jiang², Jie Yao³ and Yuanyuan Dong⁴

¹Wuhan National Laboratory for Optoelectronics, HUST,

²Department of Computer Science and Engineering, UT Arlington,

³School of Computer Science and Technology, HUST, ⁴Alibaba Group

Abstract

Popular software storage architecture Linux Multiple-Disk (MD) for parity-based RAID (e.g., RAID5 and RAID6) assigns one or more centralized worker threads to efficiently process all user requests based on multi-stage asynchronous control and global data structures, successfully exploiting characteristics of slow devices, e.g., Hard Disk Drives (HDDs). However, we observe that, with high-performance NVMe-based Solid State Drives (SSDs), even the recently added multi-worker processing mode in MD achieves only limited performance gain because of the severe lock contentions under intensive write workloads.

In this paper, we propose a novel stripe-threaded RAID architecture, StRAID, assigning a dedicated worker thread for each stripe-write (one-for-one model) to sufficiently exploit high parallelism inherent among RAID stripes, multi-core processors, and SSDs. For the notoriously performance-punishing partial-stripe writes, StRAID presents a two-phase stripe write mechanism to opportunistically aggregate stripe-associated writes to minimize write I/Os; and designs a parity cache to reduce write-induced read I/Os on parity disks. We evaluate a StRAID prototype with a variety of benchmarks and real-world traces. StRAID is demonstrated to consistently outperform MD by up to 5.8 times in write throughput without affecting the read performance.

1 Introduction

The advent of ultra-fast storage devices such as NVMe-based Solid-State Drives (SSDs) and Non-volatile Memory (NVM) with GB/s-level I/O bandwidth has dramatically narrowed the performance gap between memory and storage. Redundant Array of Inexpensive Disks (RAID) [56] can combine multiple such high-performance storage devices to further promote the overall storage performance, reliability, and capacity simultaneously. Many empirical studies [11, 19, 35] including distributed datacenter storage systems [49, 70] and enterprise storage systems [48] report that SSD drivers exhibit

reliability problems in that more than 20% of SSDs develop uncorrectable errors in a four-year period [58]. Therefore, parity-based RAIDs composed of ultra-fast SSDs have become attractive storage systems for modern data-intensive applications in supercomputing [55], big data analytics [27, 64], machine learning [8], enterprise storage [48], and cloud services [1, 32, 38, 46, 61].

HDD-based RAIDs have been extensively studied since 1988 [56]. In the literature, recent studies focus on SSD-based RAID and All-Flash-Array (AFA), with efforts to reduce SSD write-penalty by mitigating parity update [9, 16, 67], reduce garbage-collection induced performance jitter [23, 33, 42], and optimize AFA using declustering RAID approach to balance load within devices and reduce tail-latency [30, 75]. Existing RAID I/O handling techniques generally adopt a centralized stripe-processing architecture following a classic principle that trades more fast-CPU-cycles (e.g., scheduling algorithms) for fewer slow-I/Os. Nonetheless, the question of whether such RAID architecture can fully exploit the power of emerging fast storage remains unanswered.

We experimentally measure the actual performance of Multiple-Disk (MD) [45], the most popular and mature software RAID integrated into the Linux kernel for over two decades. We conduct MD running on 6 NVMe-based SSDs with 64 user threads (i.e., issuing block requests) and 64 workers threads (i.e., handling RAID stripe-writes), with the experiment environment summarized in Tables 1 and 2. The results are shown in Figure 1 (detailed in Section 3.1). With RAID0 (non-parity RAID-level), MD obtains an expected performance that approaches the aggregate raw I/O capacity of the underlying SSDs, i.e., 20GB/s and 14GB/s for read and write throughputs respectively. However, MD falls far short of the expectation in write performance in RAID5 and RAID6 (parity-based RAID-levels). Specifically, the write throughput of RAID5 is below 2.2GB/s under partial-stripe writes and below 5.2GB/s under full-stripe writes, which are only about 1/7 and 1/3 of that of RAID0, respectively. Although parity-RAIDs introduce extra parity-compute overheads, our measured XORing rate on a CPU core can reach

*Corresponding author. Email: caoqiang@hust.edu.cn

up to 29GB/s [29], which is clearly not the bottleneck.

Through profiling (detailed in Section 3.2), we experimentally uncover that the write inefficiency of parity-based RAID comes from a centralized stripe-handling architecture in the legacy MD. Specifically, a worker thread using shared data structures (e.g., stripe-list) handles write requests by efficiently collaborating with user threads, XORing threads, and device I/O threads. For HDDs and slow SSDs, this one(worker thread)-for-all(stripe) architecture utilizes fast CPU sufficiently by postponing stripe-writes to absorb more requests for reducing actual I/Os. However, a single worker thread is upper-bounded in its processing capability that fails to keep up with the fast storage. The latest MD introduces a multi-worker mechanism, referred to as the N-for-all processing model, but achieves a limited performance gain due to severe lock contention on the centralized data structures.

In this paper, we propose a novel stripe-threaded architecture, called StRAID, for parity-based RAIDs built on ultra-fast storage devices such as NVMe-based SSDs. To address the architectural drawback of the existing software RAID (MD), StRAID employs a one(worker)-for-one(stripe) model, thus significantly reducing the number of stripe-states and their lock-based checks. Furthermore, StRAID adopts a fine-grained stripe-level lock, substantially mitigating contentions on shared data structures. To tame the notoriously performance-degrading partial-stripe writes, StRAID further designs a two-phase stripe submission mechanism that opportunistically aggregates subsequent incoming writes belonging to the same stripe within a limited time window. Meanwhile, StRAID proposes a parity-block cache to speed up frequent write-induced parity reads. Fundamentally, StRAID effectively exploits stripe-based data parallelism while mitigating intra-stripe conflicts between the dedicated stripe worker thread and other threads. StRAID leverages the power of multicore CPUs that offers sufficient inexpensive-threads to fully unleash the superior IOPS provided by fast SSDs.

The main contributions of this paper are as follows.

- We experimentally observe a serious write inefficiency problem in the current MD when parity-based RAID is running on ultra-fast storage. We further reveal that the root cause is the centralized one-for-all stripe-handling architecture.
- We propose a novel parity-RAID processing architecture, StRAID, guided by a stripe-threaded one-for-one model to unleash the full performance potentials of modern hardware. We also present two key techniques, opportunistic stripe-aggregation and parity-block cache, to improve the performance of partial-stripe writes.
- We prototype and evaluate StRAID with a variety of benchmarks and real-world workloads. StRAID consistently outperforms MD by up to 5.8 times in write throughput without affecting the read performance while reducing CPU utilization.

The rest of the paper is organized as follows. Section 2 presents the background for RAID. Section 3 analyzes the performance behaviors of Linux software RAID (MD) and motivates the StRAID design. Section 4 describes StRAID's design. We evaluate StRAID in Section 5 and describe related works in Section 6. Section 7 concludes this paper.

2 Background

2.1 RAID Systems

Redundant Array of Inexpensive Disks (RAID) [56] is a classic system-level approach that combines multiple disks to improve performance, reliability and capacity simultaneously. Over the past decades, RAID has been used ubiquitously to construct and manage efficient storage servers, distributed storage [5, 54], and cloud storage [1, 38] from within and/or among storage devices.

The RAID architecture is categorized into various RAID levels based on the amount of redundancy and how redundancy is incorporated, including non-parity RAID (e.g., striping-only RAID0 and mirroring-only RAID1) and parity RAID (e.g., RAID5 and RAID6 that can tolerate one and two disk failures respectively). RAID can be implemented in either software or dedicated hardware (e.g., I/O controllers or firmware) to offer the block-addressable volume. A common N-disk RAID internally consists of multiple stripes, each of which comprises user data chunks and their corresponding parity data chunks across N disks according to an algorithmic address-mapping method. Normal reads without disk failure are directly decomposed to their constituent chunk I/Os served by the underlying disks. Normal writes in non-parity RAID behave like normal reads without accessing parity chunks.

Normal writes in parity-based RAID need extra parity generation, update, or construction operations. For a full-stripe user write where all data chunks of a stripe are written, the RAID system generates all new parity chunks at once, and then writes both data chunks and parity chunks into their corresponding disks. For a partial-stripe write where only a subset of the data chunks of a stripe are written, only after its constituent old data or parity chunks are read from the disks is the stripe updated and then written into the disks again, thus inducing numerous extra I/Os [9, 30]. This read-modify-write nature of partial-stripe writes makes them notoriously costly. When disks fail within the failure-tolerance range, the RAID transitions from its normal mode to a degraded mode to perform read, write, or resync operations.

2.2 Linux Software RAID

The Linux software RAID module, referred to as Multiple-Disk (MD) [45], is the most commonly used software RAID evolving with the Linux Kernel for over two decades. Currently, MD supports various RAID levels and RAID compositions. Non-parity-based RAID in MD perform an algorithmic

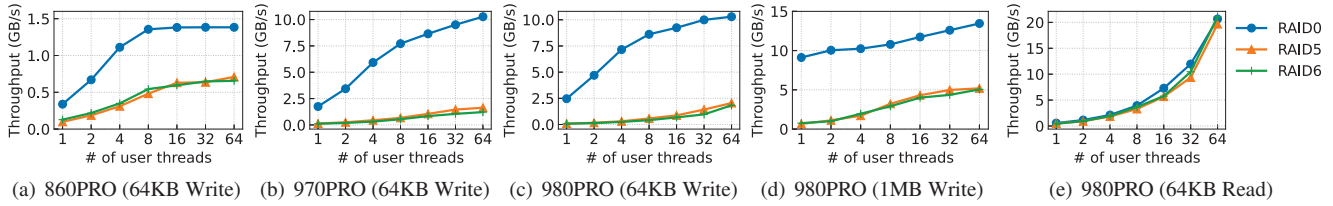


Figure 1: The throughput of Linux software RAIDs on three-types of SSDs under varying number of user threads.

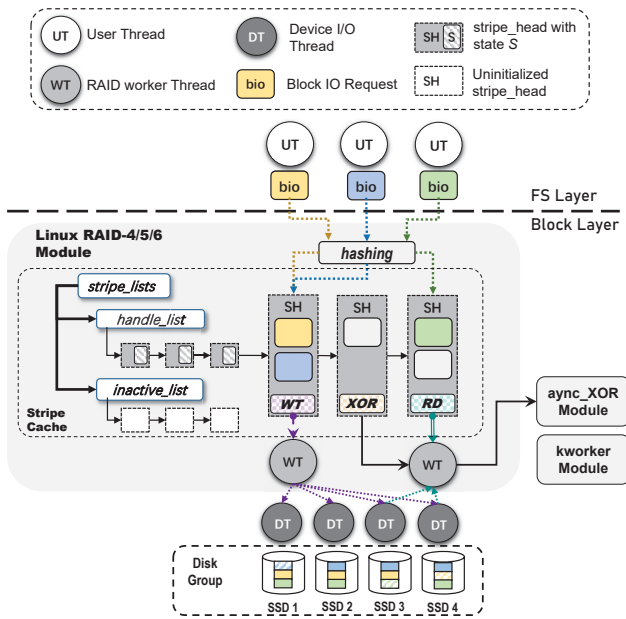


Figure 2: Architecture of Linux MD parity-based RAID.

block-to-chunk address mapping. For parity-based RAID, normal reads are similar to those in a non-parity-based RAID without parity operation. However, writes inevitably introduce several additional parity-generation/modification operations. Figure 2 shows the architecture of MD parity-based RAID and Figure 3 shows the workflow of their stripe-writes. The centralized data structure (stripe-cache) comprises inactive and handling stripe-lists, which maintain the metadata of the stripes (up to 256 by default). Each stripe has its own stripe_head containing stripe states and device states (*Devs*). *Devs* contains a set of block request structures (*bios*) pointing to their buffered pages. Specifically, a stripe and its corresponding *Devs* have 28 and 27 states respectively that are used to precisely identify the handling states of this stripe. When a stripe is processed and cleared, its corresponding stripe_head will be transferred into the inactive_list.

MD handles stripe-writes using a state machine represented as a directed acyclic graph (DAG) [17]. As shown in Figure 3, a normal user write process can be divided into 5 consecutive stages: 1) inserting/aggregating *bios* to a stripe (INS); 2) reading data/parity chunks (RD); 3) computing parity (XOR); 4) writing data/parity (WT); 5) clearing stripe (CLR). Specifically, in the first stage ①, user threads (UT) invoke *make_request()* to attach *bios* to their correspond-

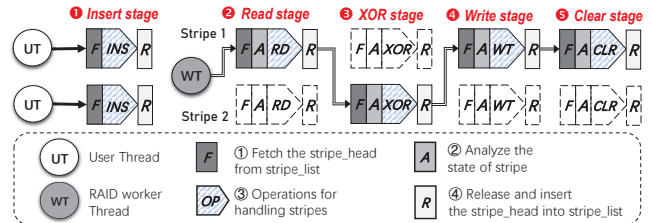


Figure 3: Stripe-write workflow of parity-based MD RAID.

ing stripe_head structures. Afterwards, a daemon worker thread (WT), i.e., *RAID5d* in MD by default, handles all active stripe_heads in a circular manner with priority.

For a full-stripe write, MD skips the second stage. For a partial-stripe write, MD must introduce write-induced reads ②, resulting in I/O amplification. More specifically, there are two stripe-updating schemes, read-modify-write (RMW) and read-construction-write (RCW) [30]. MD calculates the required number of disk-read I/Os of both RCW and RMW, selects the I/O-minimum approach, and launches the relevant disk-read I/Os. When a disk I/O thread (DT) completes the read, it sets a data-prepared flag to its *bios*. Afterwards, ③ WT checks all the involved *bios* until prepared, and then launches a parity-calculation executed by other XORing threads. When WT verifies that the parity has been prepared, ④ it invokes disk-write I/Os. ⑤ WT finally validates the completed state and clears the stripe_head. Therefore, the write process orchestrates WT, UT, and DT threads via shared-state setting and checking.

WT handles each stage of a stripe-write in four steps, as described in Figure 3: ① getting a stripe_head from a stripe_list; ② analyzing the current state of this stripe and all its involved *bios*, to determine whether this stripe is still in-flight; ③ handling the stripe by launching a given operation (e.g., XOR) through executing DAG; and ④ updating the stripe state, inserting it back into a stripe_list and selecting the next stripe. The worker thread handling a stripe exclusively accesses shared data structures and stripe-states using multiple locks. For example, in step ④, WT exclusively modifies *handle_list* with a global device lock.

For HDD-based RAID, a disk I/O takes at least several milliseconds. Therefore, a WT in Linux MD has sufficient CPU-cycles to drive all stripe-writes. With the emerging SSDs that have 2-3 orders of magnitude lower I/O latency than HDD, MD also introduces a multi-worker mechanism [39,40] that enables more numbers of functionally equivalent worker

Table 1: Evaluation Platform Specifications

Components	Configurations
Processor	Duel Socket Intel Xeon Gold 6328, 56 Cores, 128MB LLC
Memory	256GB 2666MHz DDR4
Operating System	Ubuntu 20.10 LTS with the Linux kernel version 5.13.0
MD controller	mdadm v4.1

Table 2: Characteristics of three representative SSD products.

Device Types	Device Modules	Capacity	Stable Write Thr. (MB/s)	Stable Read Thr. (MB/s)	Interfaces
SATA SSD	Samsung 860 Pro	512GB	500	510	SATA
NVMe SSD	Samsung 970 Pro	512GB	2200	3200	PCIe 3.0
NVMe SSD	Samsung 980 Pro	1TB	2600	6900	PCIe 4.0

threads to process stripes concurrently, referred to as the N-for-all processing model.

3 Analysis and Motivation

3.1 Understanding the Write Performance

Experiment Setup We start with measuring the MD performance in the RAID0, RAID5 and RAID6 levels running on three types of SSD devices, whose I/O characteristics are listed in Table 2. The platform configuration is shown in Table 1. The XORing throughput on a single CPU-core can reach up to 29GB/s. We deploy six SSD devices to construct parity-based RAID5, that is, 5+1 RAID5 and 4+2 RAID6 respectively. The chunk size in all RAID5 is set to 64KB as default. We pin each user thread (UT) to a unique CPU-core and increase the number of UTs from 1 to 64. Each UT issues random 64KB-sized writes over 30 seconds. For parity-RAIDs, we invoke up to 64 extra RAID worker threads (WT), and enlarge the stripe cache capacity from the default of 256 stripe_heads to 16K stripe_heads.

Write Inefficiency with Parity-RAID Figure 1 reports the throughput performance of MD. In all the cases, the write performance and scalability of the non-parity RAID0 far exceed those of parity-based RAID5 and RAID6. RAID0 achieves a write performance of about 1.4GB/s and 11GB/s peak throughput on 860Pro and 980Pro SSDs at 64 UTs, while RAID5 in the multi-worker mode achieves a peak write performance of lower than 0.72GB/s and 5.3GB/s, respectively. On 980Pro, the 64KB partial-stripe write throughputs of parity-RAIDs are below 2.1GB/s, which is only 1/7 of that of RAID0. Even for the 1MB full-stripe writes, the throughputs of RAID5 and RAID6 with 64 UTs are below 5.2GB/s and 5.3GB/s respectively, only about 38% of that of RAID0. It indicates that parity-RAIDs fall short of leveraging the write I/O performance of modern SSDs and the bottleneck on CPU processing is the main reason. We will show more details in the next section. Besides, normal reads of MD in all RAID levels are generally similar and scale well with the number of UTs.

We further analyze the write inefficiency of the multi-worker mechanism with RAID5 on six 980Pro SSDs. We invoke 64 UTs in either the single-worker (i.e., *Single*) mode or the multi-worker mode with the number of WTs vary-

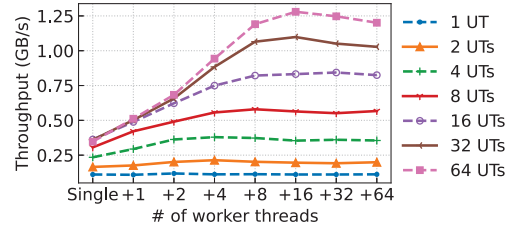


Figure 4: Write throughput of MD RAID5 under the multi-worker mechanism.

Table 3: Key function calls and locks of Linux parity-RAID.

Operations	Function as example	Description
RD/WT	generic_make_request()	Send bio to block device queues (Ⓜ in stages Ⓜ and Ⓜ)
XOR	async_xor()	Compute parity data (Ⓜ in stage Ⓜ)
F/R List	release_stripe()	Insert the stripe_head to stripe_list according to its states (Ⓜ and Ⓜ)
Lock	spin_lock_irq(device_lock)	Global MD device Lock, mainly used for updating shared structs
Analyze	analyze_stripe()	Analyze the states of a stripe and its Devs before handling (Ⓜ)
Others	-	Other software overhead

ing from 1 to 64 (i.e., +1W to +64W). Figure 4 shows that the parity-based RAID gains limited benefits from the multi-worker mode. For example, MD with 8 more WTs has a write throughput improvement of 2.4x and 3.6x over the single-worker mode under 16 and 64 UTs, respectively. However, MD’s performance gain peaks at 16 WTs, beyond which MD’s throughput starts to gradually decrease, e.g., with a 5% decline at 64 WTs. This indicates that the multi-worker mode has a diminishing return in performance beyond a relatively small number of WTs. Therefore, even in the case of 64 UTs and 64 WTs, parity-RAIDs still fall short of fully leveraging the I/O bandwidth offered by the fast SSDs.

3.2 Identifying the Root Causes

We investigate the CPU usage distribution to identify the root causes of poor write scalability of MD. We use RAID5 with fixed 64 UTs and vary the number of WTs from 1 to 64. We use perf [44] to measure CPU cycles of key functions within a WT thread, detailed in Table 3. We randomly select one WT for analysis since all WTs behave very similarly in our experiments. Figure 5 shows that CPU cycles of disk I/O (RD/WT) and XORing (XOR) decrease as the number of WTs increases, accounting for 42% of the total CPU cycles in the single-worker mode, but only 9.7% at 64 WTs. Meanwhile, the CPU cycles of stripe-write process (i.e., F/R List, Lock, Analyze and Others) increase significantly as WTs increase.

First, the global device lock (Lock) consumes a mere 4.3% of CPU-cycles in the single-worker mode but a dominant 54.6% in the 64-worker mode. As shown in Table 3, the device lock in Linux MD is spin lock, which controls concurrent accesses from WTs, UTs, and DTs to all the stripe_lists and metadata of RAID. In most cases, each WT exclusively accesses the handle_list, thus causing severe lock contention

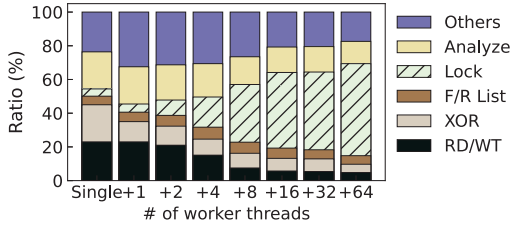


Figure 5: Breakdown of CPU cycles on key functions and locks of the worker threads in Linux MD.

among these threads. Recently, Linux Kernel contributors also found high overhead of the device lock in the read path [52] and replaced them with a lockless memory barrier, thus achieving 7x improvement in small-sized reads. However, the device lock in the write path remains a serious source of contention.

Second, checking for stripe states (*Analyze*) consumes 22% and 13.2% CPU usage in the single-worker and 64-worker modes, respectively. In Linux MD, most of the stripe states and device bio states use a set of semaphores to orchestrate UTs, WTs, and DTs. In summary, through extensive experiments, we observe that the architectural deficiency of the N-for-all centralized handling model leads to severe lock contentions due to highly-concurrent accesses to global data structures and the states of stripes.

4 Design

Given the above identified root causes of write inefficiency of MD with parity-RAIDs running on ultra-fast SSDs, we propose a stripe-threaded architecture of parity-RAID, StRAID for short. StRAID assigns a dedicated worker thread for each stripe-write, which significantly reduces lock contentions among multiple threads, and addresses the partial-stripe-write penalty with a two-phase write submission and a parity cache.

4.1 Architecture

Figure 6 illustrates the StRAID architecture for parity-RAID. StRAID does not change the data layout of the legacy MD. It persists RAID’s metadata at the pre-defined location of each disk. Each user thread (*UT*) pushes a block-write to a dedicated worker thread (*WT*) that exclusively handles its corresponding stripe. Multiple WTs process their own stripes independently, exploiting the intrinsic data parallelism among stripes. StRAID pre-allocates at least 256 WTs in the WT Pool to alleviate frequent thread creation/destroy overhead in runtime.

A normal stripe-write process in StRAID can be divided into 6 consecutive stages of ① initializing stripe_heads and inserting bios (*INS*); ② reading parity/data chunks (*RD*); ③ performing I/O batching (*BAT*); ④ computing parity (*XOR*); ⑤ writing data/parity and ⑥ clearing stripe states in SST (*CLR*). Moreover, ⑦ user threads being batched must wait for completion (*WAIT*). A notable workflow difference between StRAID in Figure 6 and the legacy MD in Figure 3 is that

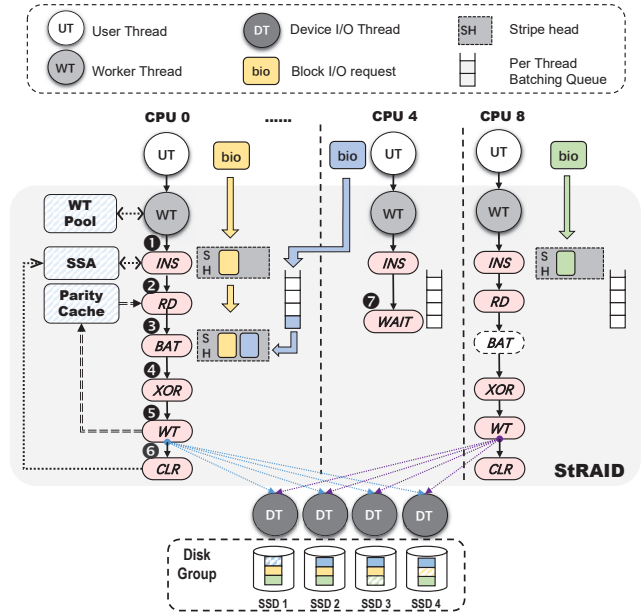


Figure 6: Architecture and process flow of StRAID

the latter’s stages of ① stripe acquisition, ② analysis and ④ stripe release are removed in the former.

Compared to the legacy MD, StRAID removes the centralized stripe_head lists and their corresponding concurrent operations. Furthermore, StRAID minimizes the number of shared stripe-states and global-state checking among WTs, because a dedicated WT handles a stripe-write exclusively. Finally, the parity computation and I/O execution processes of a stripe write are pinned to the same CPU core, thus avoiding frequent context switches and CPU cache pollution.

However, StRAID faces new challenges in effectively conducting thread collaboration and reducing the partial-write penalty. StRAID still needs a minimal shared-data structure to orchestrate UTs, WTs, and DTs in handling stripe-writes. To this end, StRAID proposes a Stripe State Table (§4.2) with lockless access features. Further, the legacy MD uses the global stripe-cache and active/passive delays to aggregate stripe-associated writes (SS-writes) that target the same stripe, thus reducing partial-write-induced disk I/Os. However, in StRAID, a user write triggers a dedicated WT to immediately and exclusively handle the corresponding stripe-write, which does not address the costly partial-write penalty. To optimize partial-stripe writes, StRAID presents a two-phase stripe submission mechanism (§4.3) to opportunistically aggregate SS-writes by employing a batching queue per WT. Further, StRAID employs a parity cache (§4.4) in memory to buffer hot parity blocks, for significantly mitigating write-induced parity-reads.

4.2 Stripe State Table

StRAID designs a Stripe State Table (SST), as shown in Figure 7, to maintain a minimal set of shared stripe-states.

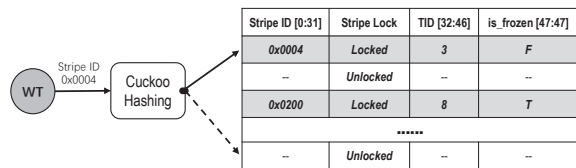


Figure 7: Stripe state table.

SST adopts a hash table to index up to 4096 active stripe-entries, each of which is handled by a dedicated worker thread. An SST-entry (48-bit) contains four fields: 32-bit *Stripe ID* uniquely specifying a stripe; 1-bit *Stripe Lock* indicating whether this stripe is currently being processed; 14-bit *TID* identifying the thread ID of the dedicated WT handling this stripe; and 1-bit *is_frozen* recording the shared stripe-state that indicates whether the stripe is allowed to batch. SST is a globally shared structure between WTs and DTs, where each entry is uniquely associated with a physical stripe and can only be exclusively modified by a WT using CAS [57] at any time. SST employs Cuckoo hashing [53] for achieving high table occupancy while preventing hash collisions. The total memory footprint of SST is smaller than 40KB.

4.3 Two-phase Stripe Submission

Partial-stripe Write Overhead A partial-stripe write causes write-induced reads and write amplification. The write-induced-read ratio (*WIRR*) and write amplification (*WA*) of RAID5 are estimated by Eq.1 and Eq.2 respectively, where *WS*, *CS* and *SS* represent write-size, chunk-size and stripe-size, respectively. When *WS* is smaller than *CS* in RAID5, a block-size write induces 2x read I/Os and 2x write amplification (one data-block write and one parity-block write) with optimal RMW strategy. As *WS* increases, the amount of write-induced-read data decreases (0 for a full-stripe write). The write amplification is larger than 1.2x on RAID5 with no disk failures.

$$Write\text{-induced-read Ratio} = \begin{cases} 2 & WS \leq CS & (RMW) \\ 1 + \frac{CS}{WS} & CS \leq WS < \frac{SS}{2} & (RMW) \\ \frac{SS}{WS} - 1 & \frac{SS}{2} \leq WS < SS & (RCW) \\ 0 & WS = SS \end{cases} \quad (1)$$

$$Write\text{ Amplification} = \begin{cases} 2 & WS \leq CS \\ 1 + \frac{CS}{WS} & WS > SS - CS \end{cases} \quad (2)$$

Existing optimizations for partial-stripe writes can be characterized by the three general approaches of write-aggregation [45], dynamic stripe size [7, 76], and parity logging [9, 10, 15, 71, 72]. The legacy Linux MD employs a global stripe-cache to absorb active user writes by postponing stripe-writes actively or passively. This write-aggregation approach reduces actual disk I/Os but increases the latency of the postponed requests, which may hurt the overall performance for low-latency SSDs. RAIDZ [7] uses a dynamic stripe size mechanism to eliminate partial-stripe writes, but assumes the support of the ZFS file system. The logging approach first persists incoming writes to an auxiliary fast-disk (e.g., SSD

or NVM), and then rewrites the relevant stripes to original locations in the background, thus leading to at least 2x write amplification and bottlenecks from log-devices.

Two-phase Stripe Submission Without a global stripe-cache, StRAID designs a two-phase stripe submission mechanism to opportunistically absorb SS-writes. StRAID divides the stripe-write process into two phases: a batching phase and a frozen phase. Specifically, Figure 8 (referred by circled numbers) and Algorithm 1 (referred by line numbers) describe the two-phase submission using an example where three concurrent I/O threads issue requests targeting the same stripe (*SI*). A worker thread 1 (*WT1*) receives bios from its corresponding UT, and acquires a stripe lock to begin stripe processing (*Time* ①, *line* 2) by CAS operation. *WT1* first initializes the stripe states in SST, then it determines the reconstruction method (RCW/RMW) for this stripe and reads the required parity/data blocks from the disk (*line* 4-6). Shortly after *WT1*'s arrival, a second worker thread (*WT2*) arrives and seeks SST, only to find that the targeted stripe is locked but enables batching (*Time* ②, *line* 14). It inserts bios belonging to this stripe to the batching queue of the handling thread *WT1* (*Time* ③, *line* 15) and then suspends itself.

When *WT1* completes its batching phase, it immediately transitions the stripe into the frozen phase (*Time* ④, *line* 7) by using the CAS operation. At this point, the stripe is not allowed to accept new bios. Hence, the newly arrived bios from worker thread 3 (*WT3*) (*Time* ⑤) are blocked and have to wait for the stripe write's completion. *WT1* coalesces all requests in its batching queue and processes them as a whole, then it re-executes parity read (if required) in accordance with the aggregated stripe-write, and performs XORing and data/parity writes to reconstruct the stripe. Finally, *WT1* clears up the stripe states of *SI* in SST and releases the stripe lock. The corresponding waiting thread *WT2* will also return successfully (*Time* ⑥, *line* 12). Next, *WT3* successively acquires the *Stripe Lock* to handle its requests on the stripe.

In contrast to the stripe-cache approach for aggregating SS-writes used in Linux MD, the novelty of the two-phase writing approach in StRAID leverages the latency of executing a reconstruction read in the batching phase to opportunistically aggregate incoming SS-writes. It ensures the efficiency of each handling thread, thus achieving better throughput without sacrificing I/O latency.

4.4 Parity Cache

Partial-stripe writes induce frequent parity-block accesses and cause performance degradation. To alleviate this problem, StRAID further designs a parity cache to keep hot parity-blocks in the memory to reduce disk I/Os. Previous works [9, 63, 72] use the logging approach to absorb parity updates at the cost of write amplification and potential bottlenecks at the log-devices. StRAID, instead, uses the parity cache only for eliminating parity reads induced by partial-stripe writes.

Algorithm 1 Two-phase stripe submission

```

1: while all bios are handled do
2:   if get_stripe_lock(stripe_id) then
3:     init_SST(stripe_id)
4:     Determine reconstruction method
5:     if is_partial-stripe write then
6:       read from disks
7:       set is_frozen = true in SST and pull batching bios from queue
8:       if Data is not enough for reconstruction then
9:         Re-read from disks
10:      Compute XOR and reconstruct stripe
11:      clear_SST(stripe_id)
12:      release_stripe_lock(stripe_id)
13:   else
14:     if !is_frozen(stripe_id) then
15:       insert bio to queue with TID
16:     else
17:       handled = false
18:       continue
19:   Waiting for all bios to complete

```

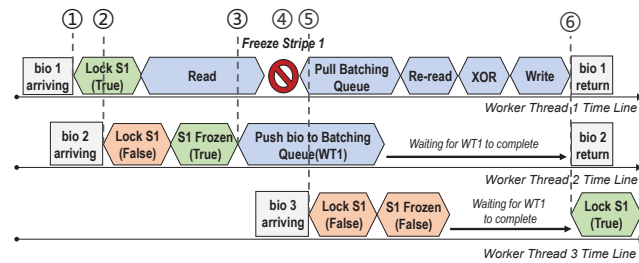


Figure 8: Workflow of two-phase stripe submission, an example with 3 concurrent worker threads (*WT1-WT3*) targeting the same stripe.

The stripe parity has higher access-frequency than its stripe data for partial stripe writes. Therefore, caching parity data is significantly valuable.

The architecture of parity cache is shown in Figure 9. It contains a high concurrency hash table with $O(1)$ lookup cost, and uses an LRU-based cache replacement policy to capture frequently accessed parity data. Each hash entry corresponds to a physical stripe and contains three fields, *Stripe_ID*, *p_data* and *q_data* with the latter two pointing to cached parity data aligned with the 4KB block size. RAID5 only uses the *p_data* pointer and RAID6 uses both pointers. In addition, each entry has a fine-grained read-write lock for synchronization.

The cache module updates and queries at the block granularity, but inserts and deletes hash entries at the stripe granularity. For updates and queries, a WT needs to first insert the *Stripe_ID* into the LRU and searches the relevant entry. The read lock is required to access cached data because each stripe can only be updated exclusively by one WT. During the stripe-write process, a WT first searches the parity cache and acquires hit parity blocks. When missed, WT updates both disks and cache in a write-through scheme with the newest parity data after XORing.

StRAID periodically triggers a dedicated thread to clean up the stripe in the background. When the cache size exceeds

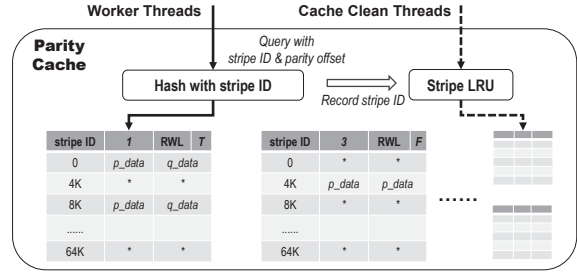


Figure 9: Architecture of the parity cache.

a threshold (64MB capacity for 16K parity blocks as default), the cleaning thread removes the entries of evicted stripes from the cache according to the LRU policy.

4.5 Recovery and Degraded Mode

Crash Consistency and Recovery After a system crash, part of the chunk writes belonging to a stripe-write may be lost, making the stripe inconsistent between its data and parity. StRAID uses a bitmap to record the current update-state of each chunk. Compared with Linux MD, StRAID's bitmap has basically the same data structure and layout, but can only be updated and flushed by dedicated threads. For each chunk update, StRAID first sets the corresponding bitmap bits and changes their involved memory-page as dirty, then flushes the page to the underlying SSDs via the memory mapping mechanism. The bits will be cleared after their corresponding chunks are written to the disk. Similar to MD, StRAID groups bitmap updates in a batch to avoid frequent disk I/Os. In the experiment, it is found that flushing the bitmap only incurs a very small overhead (less than 2%) when handling stripe writes. With unexpected power failures, StRAID will fetch the bitmap from the disks and restore it to the consistent state after reboot. Moreover, StRAID has the option to use a journal device [3] as a writeback cache to prevent the write hole problem [22].

Resync and Degraded Mode StRAID supports degraded reads, degraded writes and resync operations in the same way as the legacy MD because the underlying data layout is identical. For stripe writes, StRAID identifies the degraded stripe and handles it after entering the frozen phase. The resync operation reads all the data blocks from disks and compares their calculated parity results with their on-disk parity data. It is triggered upon RAID initialization, or reconstruction from disk replacement. We evaluate the performance of StRAID in degraded mode in Section 5.

5 Evaluation

5.1 Evaluation Setup

Platform We run all experiments on a server (detailed settings listed in Table 1) and three types of SSD devices (described in Table 2). The CPU-core can reach 29GB/s XORing

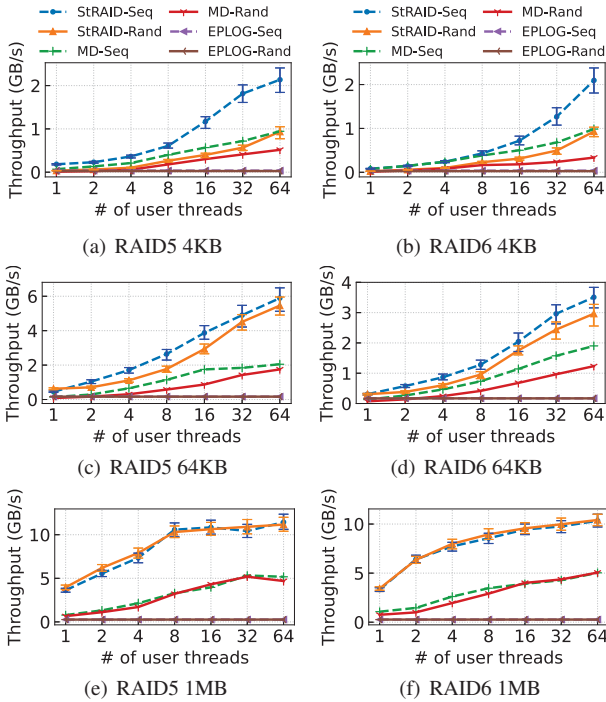


Figure 10: Write scalability on three different RAID systems.

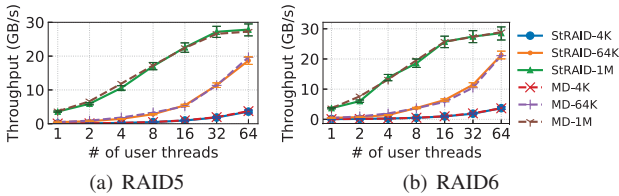


Figure 11: Read scalability of StRAID and MD.

throughput and the PCIe I/O bandwidth is 48GB/s [21], exceeding the aggregate sequential bandwidth of 6 NVMe-based SSDs (2.6GB/s stable write throughput per SSD, 15.6GB/s in total). In our experiments, we bind all the I/O threads and worker threads to the same CPU socket-0 to avoid remote accesses of memory and PCIe, i.e., the NUMA issues.

RAID systems setup We evaluate StRAID and Linux MD (MD) of the RAID5 (5+1) and RAID6 (4+2) levels built on 6 SSDs. The chunk size is set to 64KB by default. StRAID has a 64MB-sized parity cache. Linux MD has a 16K-entry stripe-cache and up to 64 worker threads. This is a setting that MD is shown by our experiments to achieve the best throughput. In addition, we compare StRAID with EPLOG [9] that mitigates parity update overhead by redirecting parity traffic to separate dedicated HDD logging devices. To prevent the log-devices from becoming a bottleneck, we replace the HDDs of EPLOG with the same type of SSDs used in the main RAID array.

Workloads We implement a program to issue direct block I/O requests with sequential or random access patterns as micro-benchmark. We run each experiment ten times and take the average as the results. We further select six representative block traces summarized in Table 4 as trace-driven

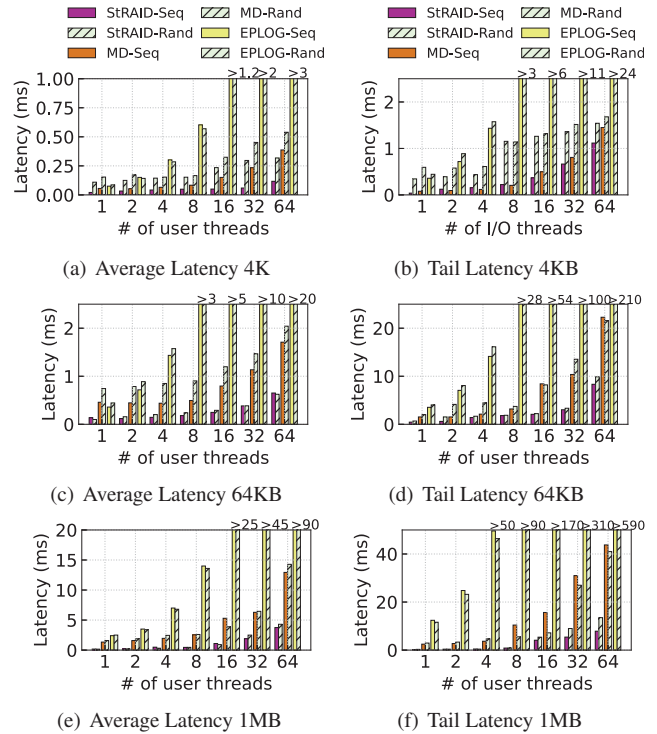


Figure 12: Average and tail latency of RAID systems.

macro-benchmarks. We implement a trace player in C++ using POSIX sync to generate direct block I/O requests to the underlying RAID systems.

5.2 Micro-benchmark

We measure the write throughput, average and tail latency, and amount of disk read/written data on MD, StRAID and EPLOG with RAID5 and RAID6 on 980Pro SSDs, respectively. We generate workloads with a different number of concurrent I/O-issuing threads (i.e., UTs) and varying access patterns. Three default I/O sizes are: 4KB (default block-size of file systems, page cache, and block devices), 64KB (partial-stripe write size), and 1MB (full-stripe write size).

Throughput Figure 10 reports the write throughput of StRAID, MD and EPLOG in RAID5 and RAID6, respectively. The errorbars are added on StRAID’s results. The throughput of StRAID exceeds that of MD and EPLOG respectively by up to 2.1x and 1.4x with 4KB-sized writes and 1.5x and 1.3x with 64KB-sized writes with a single UT, respectively. This is because StRAID effectively reduces the overhead of handling stripe-states. As the number of UTs increases to 64, StRAID achieves up to 2.0GB/s±0.2GB/s and 6.0GB/s±0.8GB/s peak throughput with 4KB and 64KB writes respectively, representing 2.1x/59.1x and 2.9x/35.1x performance improvement over MD/EPLOG. In addition, EPLOG achieves 1.4x-1.9x higher throughput than MD under random write at a single UT, because it avoids partial-write-induced reads with parity-logging. However, EPLOG does not scale at all with more

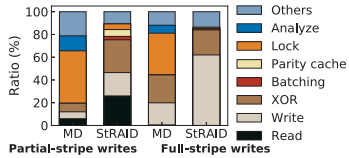


Figure 13: Breakdowns of CPU cycles on StRAID and MD

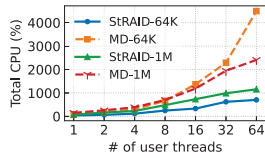
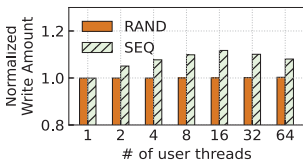
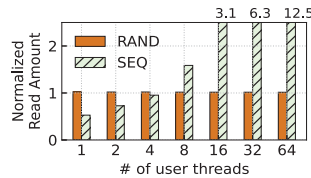


Figure 14: Total CPU utilizations



(a) Written data



(b) Read data

Figure 15: Amount of data written to and read from disks by StRAID, normalized to that of MD.

UTs, because it uses a global lock for serializing each write operation.

For full-stripe writes (i.e., 1MB), StRAID achieves 4.6x/12x and 5.2x/13x higher write throughput in random and sequential cases than MD/EPLOG with a single UT, respectively. As the number of UTs increases to 8, StRAID’s throughput saturates the device bandwidth, with an almost fixed increase of about 6GB/s over MD. With 64 UTs, the peak write throughput reaches 11.4GB/s±1.1GB/s and 10.4GB/s±1.0GB/s in StRAID under RAID5 and RAID6 respectively, which are 2.1x higher than those of MD (5.2GB/s and 5.1GB/s) and 41x higher than EPLOG (0.25GB/s and 0.26GB/s). StRAID’s full-stripe writes nearly unleash the full power of the SSD performance, while MD suffers from heavy contention on the global data structures.

Moreover, Figure 11 shows the read throughput of StRAID and MD with varying-size reads. The average read throughput difference between MD and StRAID is less than 5% in RAID5 and RAID6 respectively, demonstrating that StRAID does not affect read performance.

Latency and Breakdown of CPU-cycles Figure 12 shows the average and tail (99th-percentile) latency under RAID5 in StRAID, MD and EPLOG, respectively. StRAID significantly outperforms MD and EPLOG in both average and tail latencies performance under 64 UTs, reducing latency by 75% and 98.2% with 4KB block-writes, 76% and 97.1% with 64KB partial-stripe writes, and by 69% and 95.2% with full-stripe writes. StRAID reduces 22%-67% tail latencies from MD under 64 UTs. The tail latency of EPLOG is 6.5x-42.1x higher than StRAID under multiple UTs, since the global lock in EPLOG makes its average and tail latencies much higher than those of MD and StRAID.

To better understand the reasons behind StRAID’s superiority, Figure 13 shows the breakdown of the CPU-cycles of key functions consumed by MD and StRAID with 64 UTs issuing random writes, respectively. For partial-stripe writes, the combined CPU-cycles on XORing and disk I/Os account

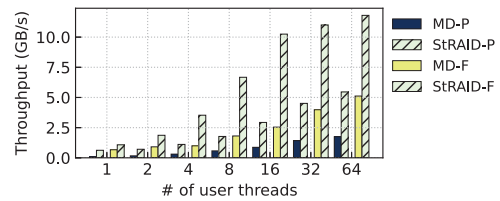


Figure 16: StRAID and MD throughput with partial-stripe (*-P) and full-stripe writes (*-F) of different chunk sizes.

Table 4: Characteristics of block I/O traces used in the macro-benchmark evaluations

Trace	Write Ops (millions)	Data Written (GB)	Avg.write size (KB)	Read Ops (millions)	Data Read (GB)	Avg.read size (KB)
Pangu-A	1.89	113.24	63.21	0.24	4.06	17.99
Pangu-B	2.44	81.32	35.08	0.30	18.61	65.24
prxy_0	12.14	53.80	4.65	0.38	3.05	8.33
prn_0	4.98	45.97	9.67	0.60	13.12	22.84
varmail	3.39	39.20	12.13	0.05	5.38	114.05
filesrver	1.19	99.45	87.56	0.47	42.37	95.49

for 76% of the total CPU usage in StRAID, while that of MD is less than 20%. The average stripe-write handling overhead of StRAID, i.e., 60μs, is about 20 times less than that of MD, i.e., 1180μs. Besides, the lock overhead on StRAID and MD account for 5.1% and 46.1% of the total CPU usage, respectively. StRAID efficiently mitigates lock contentions through the stripe-threaded architecture and the lockless access features in SST.

For full-stripe writes, the lock, XORing and I/O-write of StRAID account for 1.3%, 22.5% and 62.6% of the total CPU usage, respectively, in contrast to their MD counterparts of 36.7%, 24.5% and 19.4%, suggesting that StRAID achieves to make better advantage of SSDs’ high write bandwidth. In addition, the two-phase submission and the parity caching use only 6% and 1.5% of the stripe-write CPU-cycles of partial-stripe and full-stripe writes, respectively.

CPU utilization We compare the CPU utilizations of StRAID and MD under random full-stripe and partial-stripe write workloads respectively, with the same RAID5 settings in Figure 10. Results in Figure 14 show that the total CPU utilization of MD is up to 6.3x higher than StRAID with 64 UTs. Even when the number of UTs is less than 8, the CPU usage of MD is 2x higher than that of StRAID on average. Combining with the throughput results shown in Figure 10, MD with 4495% CPU-core utilization consumes only 1/3 of the SSDs bandwidth, in contrast to StRAID that consumes 86.9% of the SSDs bandwidth with 1156% CPU-core utilization.

Moreover, 64KB-sized partial-stripe writes of MD (*MD-64K*) consume up to 80% more CPU than full-stripe writes (*MD-1M*) with 64 UTs. MD’s inefficiency stems from its high consumption of CPU cycles required to handle in-flight partial-stripe writes. On the contrary, StRAID-64K consumes only 25% less CPU cycles than StRAID-1M because StRAID gains higher throughput for full-stripe writes that consumes more CPU resources for computing XOR and issuing I/Os.

Read/Write amplification Figure 15 shows the amount of

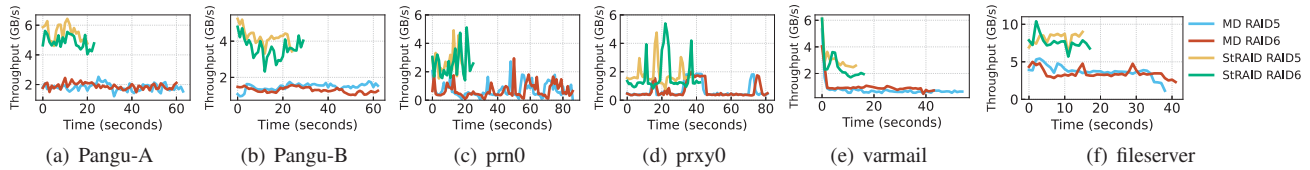


Figure 17: Throughput of StRAID and MD on trace-driven workloads.

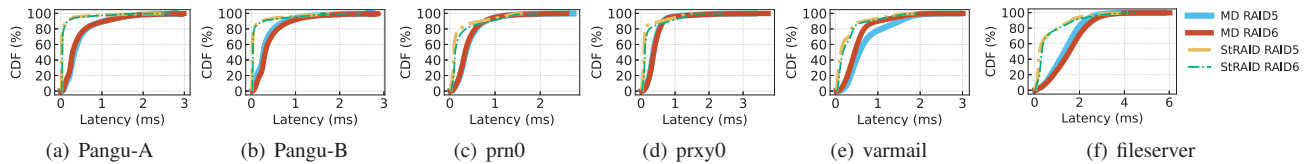


Figure 18: Latency CDF of StRAID and MD on trace-driven workloads.

data written to and read from disks by StRAID in RAID5, normalized to that of MD. For random writes, StRAID and MD have exactly the same amount of data written and data read because stripe-write aggregation is rare for random writes. For sequential writes, however, the amount of data written in StRAID is up to 12% larger than MD. This is because the two-phase submission in StRAID has a smaller aggregation window (i.e., proportional to SSD-read latency), which is slightly less efficient than Linux MD’s active delays of sequential stripe writes.

For sequential writes, the amount of data read in StRAID and MD varies significantly with different numbers of UTs. With a single UT, the amount of read data in StRAID is 1/2 of that in MD, because the parity caching mechanism effectively reduces parity reads. However, with 64 UTs as the worst case, StRAID reads 12x more data than MD. It is because MD postpones and aggregates almost all stripe-associated writes (SS-writes) into full-stripe writes, thus reducing the amount of write-induced-read data (e.g., 1.1% of user-written data). In contrast, a StRAID’s worker thread immediately executes reading parity/data chunks before performing write-aggregation, resulting in a nearly fixed number of write-induced reads (e.g., 13.7% of user-written data). However, since the high read IOPS of fast SSDs can completely absorb the increased number of read I/Os, StRAID’s write performance can still be 5.2x higher than MD.

Chunk Sizes We evaluate the effect of chunk size configuration on the performance of StRAID and MD with 64KB-sized writes in RAID6. The chunk size is set to 8KB for the full-stripe-write case (*StRAID-F* and *MD-F*), and 64KB for the partial-stripe-write case (*StRAID-P* and *MD-P*), respectively. Figure 16 shows that both StRAID and MD benefit significantly from full-stripe write workloads. The throughput of StRAID-F reaches 11.8GB/s with 64 UTs, about 1.9x higher than StRAID-P. Similarly, the throughput of MD-F is up to 3.1x higher than MD-P. However, the peak throughput of MD-F (8KB chunk size and 64KB write size) remains at 5.3GB/s, consistent with the results shown in Figure 1(d) (with 64KB chunk size and 1MB I/O size). It indicates that the peak throughput of StRAID is sensitive to the chunk size

setting. An insight from this experiment is that it is beneficial to set StRAID’s chunk size smaller, such as 8KB, to take full advantage of full-stripe writes.

5.3 Macro-benchmark

We use six representative block traces from Filebench [60], cloud-based application traces from Alibaba-Pangu [47] and Microsoft [51] to evaluate StRAID’s performance. Table 4 summarizes the characteristics of these workloads, most of which are read-write mixed or write-dominated. In the experiments, we enable 32 WTs in MD and StRAID, and evaluate them in both the RAID5 and RAID6 levels with a chunk size of 8KB. We invoke 32 user threads to replay these traces continuously, mimicking high-intensity workloads.

Figure 17 shows the throughput of StRAID and MD over time. StRAID achieves up to 2.8x higher throughput than MD, and shortens the total running time by an average of 64% across 6 workloads. In the fileserver workload, StRAID achieves peak and average throughput of 10.3GB/s and 7.9GB/s respectively, in contrast to their MD counterparts of 5.0GB/s and 3.2GB/s. The fileserver workload has the largest average write size, so that StRAID benefits from full-stripe writes. For the cloud-based workloads, StRAID’s average throughput is 3.1x and 3x higher than MD’s in Pangu-A and Pangu-B, respectively. The prxy0 workload exhibits the lowest average throughputs among all workloads, 1.8GB/s and 0.6GB/s for StRAID and MD respectively. This is because the prxy0 trace has the smallest average write size (i.e., 4.6KB) among all workloads, leading to a large amount of partial-stripe writes for both StRAID and MD. Further, it is observed that StRAID in RAID5 is 10%-15% better than in RAID6 among all the workloads, because RAID5 has less parity data than RAID6.

Figure 18 shows the latency CDFs of StRAID and MD across all the workloads. StRAID shows significantly better CDF profiles, with about 80% and 69% lower average latency than MD in workloads Pangu-A and Pangu-B, respectively. For the other four workloads, StRAID also has at least 49% lower average latency than MD. The median latencies of

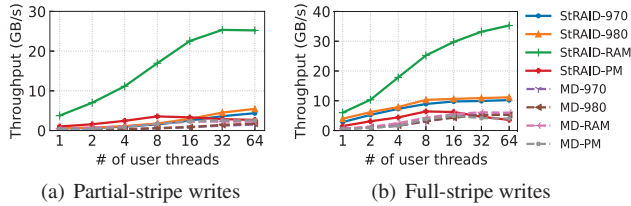


Figure 19: Performances of StRAID and MD on different SSDs and RAMs.

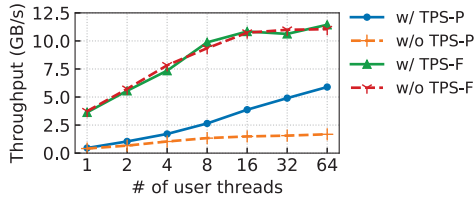


Figure 20: StRAID throughput with partial-stripe (*-P) and full-stripe writes (*-F) when running with/without two-phase stripe submission (w/ TPS and w/o TPS).

StRAID in workloads Pangu-A, Pangu-B and filebench are almost ten times lower than MD, while for workloads varmail, prn0, and prxy0 StRAID’s is 78%, 74% and 75% lower than MD’s respectively. The 99th-percentile tail latency in StRAID is 25% lower than that in MD among all workloads on average. For example, StRAID’s tail latency is up to 31.1% and 31.9% lower in workloads Pangu-A and prn0. StRAID’s advantage over MD in tail latency is lower than in average latency, because the high tail latency of both StRAID and MD mainly comes from write latency spikes caused by internal maintenance operations (e.g., garbage collection) within the SSD devices.

5.4 Sensitivity Study

Experiment with other devices Next, we evaluate the sensitivity of StRAID and MD to different types of storage devices. We first build StRAID and Linux MD on six Intel Optane PMs (in AppDirect Mode) [28] and lower-performance 970Pro SSDs, and compare these performances with that in 980Pros. The raw read and write bandwidth per PM can reach 6GB/s and 2GB/s [74], respectively. Further, we test the extreme RAID performance over six ramdisks on 128GB DRAM. We invoke up to 64 UTs with 64KB write-size for partial-stripe write-load and 1MB for full-stripe write-load.

Results in Figure 19 show that StRAID on 980Pro SSDs exhibits up to 20% higher throughput than it on 970PRO SSDs. In contrast, the performance difference of Linux MD on these two different types of SSDs is less than 5%. The throughput of StRAID on PMs is up to 50% and 35% higher than MD on partial-stripe and full-stripe writes, respectively. We also find that StRAID on PMs shows up to 26% higher throughput in partial-stripe writes than that SSDs. This is because PM has one order of magnitude lower read latency than SSDs, thus StRAID could handle stripes more efficiently.

Table 5: Parity cache capacity

	Written Data (GB)	Read Data (GB)	Cache Hit Rate	Average Thr. (MB/s)
StRAID-NC	54.10	13.64	-	1482
StRAID-4M	54.10	12.12	0.39	1593
StRAID-16M	54.10	12.01	0.45	1636
StRAID-64M	54.10	11.75	0.55	1711
StRAID-256M	54.10	11.23	0.57	1726
MD	53.80	6.10	-	667

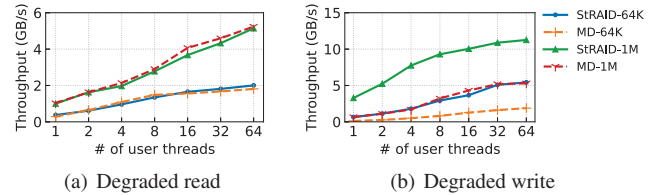


Figure 21: Read and write performance on degraded StRAID and Linux MD.

Moreover, StRAID on PM shows a throughput drop at higher than 8 UTs, because PM has a limited concurrency [74].

In addition, StRAID on RAMs delivers up to 5.8x higher write throughput than MD. At 64 UTs, StRAID reaches up to 35.2GB/s random write throughput and 32.7GB/s sequential write throughput, respectively, in contrast to their MD counterparts of 5.8GB/s and 5.7GB/s. It shows that StRAID has the potential to effectively exploit faster storage like the emerging PCIe 5.0 SSDs [18] in the near future.

Two-phase Submission We analyze the performance contributions of the two-phase stripe submission (TPS) mechanism of StRAID in RAID5. We run the experiment with (w/ TPS) and without (w/o TPS) two-phase submission, respectively. The request size is set to 1MB for the full-stripe-write case (*-F), and 64KB for the partial-stripe-write case (*-P). We issue requests with sequential access patterns. Figure 20 shows that StRAID with TPS achieves 3.5x improvement of average throughput than without TPS for partial-stripe writes at 64 UTs. The two-phase submission allows request aggregation on writes belonging to a same stripe and handles them in a batch. StRAID without TPS, by contrast, has to individually execute each writes on a stripe. Besides, the performance contribution of two-phase submission on full-stripe write load is less than 4%, because the requests targeting different stripes will not be aggregated.

Parity Cache We analyze the effectiveness of the parity cache on StRAID performance. We compare the StRAID with the parity cache disabled (StRAID_NC) against the StRAID with its parity cache capacity varying from 4MB to 256MB (StRAID_4M to StRAID_256M). We select the RAID5 level with 8KB chunk size as an example. We run the prxy0 workload with the most partial-stripe writes among all workloads tested, which has 9.5% write operations (0.51 million ops) with I/O sizes of 40KB or larger (i.e., full-stripe write size for 8KB chunk size), and the sequential write ratio is 66% (8.02 million ops). We measure the average throughput, cache hit rate, and amount of disk read/written data, respectively.

Results are shown in Table 5. As the parity cache capacity increases, the cache hit ratio increases from 39% at StRAID-4M to 57% at StRAID-64M, resulting in a 16% improvement of average throughput. In addition, increasing the cache capacity to beyond 64MB contributes less than a 5% increase in both the cache hit ratio and throughput, because the cache has captured most of the parity data under such conditions.

5.5 Resync and Degraded Mode

We assess the performance of StRAID and Linux MD in the degraded mode under RAID5. One random SSD in the RAID array is set as failed. Then, a varying number of UTs issue reads and writes of 64KB and 1MB size, respectively. Results in Figure 21 show that the read throughput of degraded StRAID and Linux MD is almost the same, with an average difference of less than 5%. Meanwhile, the write performance of degraded StRAID is 50-70% higher than that in Linux MD with multiple UTs. This is because the processing flow of write operation in degraded mode is basically the same as that in the normal mode. In addition, StRAID and MD apply the same resync approach.

6 Related Works

SSD-aware RAID SSD-based RAID systems have been extensively studied and can be roughly classified into three groups: 1) taming tail-latency by alleviating GC impact [33,68,69,73]; 2) enhancing data reliability by optimizing parity distribution or conducting wear leveling across SSDs [4,41,65]; and 3) mitigating the overhead of parity writes [9,15,26,31,72]. StRAID focuses on the multi-threaded processing architecture in RAID systems and can complement these works.

All-Flash-Array Systems RAID for AFA (all-flash-array) systems have been studied for RAID data layout optimization [50,75] and taming tail-latency by alleviating GC impact [33,59]. FusionRAID [30] improves the latency performance of the RAID system for SSD pools by leveraging the Latin-square-based deterministic addressing methods proposed in RAID+ [75], while proposing an out-of-place write method for optimizing parity-updates. SWAN [33] tames tail-latency by alleviating SSD GC impact in an all-flash-array system. Complementary to them, StRAID focuses on the stripe-write process on multi-core processors and fast SSDs without any modification of the RAID data layout. Therefore, StRAID as a new stripe-handling engine can be used in AFA systems to exploit modern hardware with high internal parallelism.

Parity Write Optimization The stripe aggregation method is widely studied to construct full-stripe writes for reducing the write-induced reads or reducing the number of parity writes to SSDs. Previous works [15,16,26,63,72] use an NVRAM or SSD as a cache to absorb incoming write data and/or parity information and delay parity updates with extra devices. In contrast, the parity cache in StRAID is located in memory and only used to accelerate read I/Os for stripe

reconstructions. Existing systems [20,30,67] first steer all writes to a logging zone and then write back to the RAID zone in the background. Such an aggregation approach requires additional storage and double the amount of data written to SSDs. In comparison to these efforts, StRAID performs an in-place update for stripe writes and opportunistically aggregates writes without extra storage requirements.

Block IO Scheduling Prior studies on block IO scheduling are focused on optimizing multi-queue management including prioritization [37], fairness queuing [24,77], policy-based storage provisioning and management [2,62] and providing low scheduling latency [25]. StRAID is a RAID stripe-write engine on top of and thus complementary to these block IO scheduling approaches. Additionally, compared with other RAID systems that adopt FTL-level block I/O scheduling [34,66,78], StRAID considers SSDs as black boxes, making it highly portable and non-intrusive.

Multicore Optimization Previous studies have addressed the scalability issues in key-value stores [12,13], file systems [6,14,43], volume management [36] and block drivers [25] with multicore processors and high-performance devices (e.g., SSDs and NVMe). MAX [43] demonstrates that lock contentions are the major reasons for poor scalability in file systems. These works exploit the potentials of parallelism on multicore processors and fast SSDs through localized key data structures and fine-grained lock designs. The Linux kernel contributors optimize lock mechanisms to improve read performance [52]. In this paper, StRAID focuses on optimizing the write path of the MD parity-RAID architecture and addresses the software overhead in handling stripe writes.

7 Conclusion

We experimentally reveal that Linux MD with parity-based RAID systems cannot fully exploit the potentials offered by high-performance SSDs due to the architectural drawback of centralized stripe-writes. We propose a stripe-threaded parity-RAID (StRAID) to efficiently handle stripe-writes in parallel. StRAID introduces a two-phase stripe submission mechanism for aggregating partial-stripe writes and a parity cache for hot parity-accesses. Through extensive trace-driven evaluations, StRAID is shown to significantly and consistently outperform MD parity-based RAID in write performance without sacrificing read performance.

Acknowledgments

We would like to thank the anonymous shepherd and reviewers for their valuable feedback and suggestion. This work was supported in part by NSFC No. 62172175, Creative Research Group Project of NSFC No. 61821003, National key research and development program of China under Grant 2018YFA0701800, the US National Science Foundation Grant CNS-2008835, and Alibaba Innovative Research.

References

- [1] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: a case for cloud storage diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 229–240, 2010.
- [2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 775–787. USENIX Association, 2018.
- [3] Apache. A journal for md/raid5. 2021. <https://lwn.net/Articles/665299/>.
- [4] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential RAID: rethinking RAID for SSD reliability. In Christine Morin and Gilles Muller, editors, *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, pages 15–26. ACM, 2010.
- [5] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. Finding a needle in haystack: Facebook’s photo storage. In *OSDI*, volume 10, pages 1–8, 2010.
- [6] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 69–86, 2017.
- [7] J. Bonwick and B. Moorei. Zfs: The last word in file systems. <http://opensolaris.org/os/community/zfs/docs/zfslast.pdf>.
- [8] John F. Canny, Huasha Zhao, Bobby Jaros, Ye Chen, and Jiangchang Mao. Machine learning at the limit. In *2015 IEEE International Conference on Big Data (IEEE BigData 2015), Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 233–242. IEEE Computer Society, 2015.
- [9] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. Elastic parity logging for SSD RAID arrays: Design, analysis, and implementation. *IEEE Trans. Parallel Distributed Syst.*, 29(10):2241–2253, 2018.
- [10] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. Elastic parity logging for SSD RAID arrays: Design, analysis, and implementation. *IEEE Trans. Parallel Distributed Syst.*, 29(10):2241–2253, 2018.
- [11] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In Gregory R. Ganger and John Wilkes, editors, *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*, pages 77–90. USENIX, 2011.
- [12] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. Spandb: A fast, cost-effective lsm-tree based KV store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 17–32, 2021.
- [13] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS ’20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1077–1091. ACM, 2020.
- [14] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Scalable persistent memory file system with kernel-userspace collaboration. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 81–95, 2021.
- [15] Ching-Che Chung and Hao-Hsiang Hsu. Partial parity cache and data cache management method to improve the performance of an ssd-based RAID. *IEEE Trans. Very Large Scale Integr. Syst.*, 22(7):1470–1480, 2014.
- [16] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1683–1694, 2015.
- [17] W.V. Courtright, G. Gibson, M. Holland, and J. Zelenka. A structured approach to redundant disk array implementation. In *Proceedings of IEEE International Computer Performance and Dependability Symposium*, pages 11–20, 1996.
- [18] Samsung Electronics. Samsung develops high-performance pcie 5.0 ssd for enterprise servers.

<https://www.samsungsemiconstory.com/global/samsung-develops-high-performance-pcie-5-0-ssd-for-enterprise-servers/>.

- [19] Nima Elyasi, Mohammad Arjomand, Anand Sivasubramaniam, Mahmut T. Kandemir, Chita R. Das, and Myoungsoo Jung. Exploiting intra-request slack to improve SSD performance. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 375–388. ACM, 2017.
- [20] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth Gibson. Diskreduce: Replication as a prelude to erasure coding in data-intensive scalable computing. *SC11*, 2011.
- [21] Dean Gonzales. Pci express 4.0 electrical previews. In *PCI-SIG Developers Conference*, 2015.
- [22] Matthias Grawinkel, Lars Nagel, and André Brinkmann. Lonestar raid: Massive array of offline disks for archival systems. *ACM Transactions on Storage (TOS)*, 12(1):1–29, 2016.
- [23] Mingzhe Hao, Gokul Soundararajan, Deepak R. Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The tail at store: A revelation from millions of hours of disk and SSD deployments. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, pages 263–276, 2016.
- [24] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. Multi-queue fair queuing. In Dahlia Malkhi and Dan Tsafir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 301–314. USENIX Association, 2019.
- [25] Jaehyun Hwang, Midhul Vuppapapati, Simon Peter, and Rachit Agarwal. Rearchitecting linux storage stack for μ s latency and high throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 113–128, 2021.
- [26] Soojun Im and Dongkun Shin. Flash-aware RAID techniques for dependable and high-performance flash memory SSD. *IEEE Trans. Computers*, 60(1):80–92, 2011.
- [27] NETAPP INC. Data ontap 8. 2010. <http://www.netapp.com/us/products/platform-os/data-ontap-8/>.
- [28] Intel. Intel optane dc persistent memory. <https://www.intel.com/content/www/us/en/architecture-and-technology>.
- [29] Intel. Isa-l performance report. <https://01.org/intel%2%AE-storage-acceleration-library-open-source-version/documentation/documentation>.
- [30] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. Fusionraid: Achieving consistent low latency for commodity SSD arrays. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 355–370, 2021.
- [31] Chao Jin, Dan Feng, Hong Jiang, and Lei Tian. RAID6L: A log-assisted RAID6 storage architecture with improved write performance. In *IEEE 27th Symposium on Mass Storage Systems and Technologies, MSST 2011, Denver, Colorado, USA, May 23-27, 2011*, pages 1–6, 2011.
- [32] Ram Kesavan, Jason Hennessey, Richard Jernigan, Peter Macko, Keith A. Smith, Daniel Tennant, and Bharadwaj V. R. Flexgroup volumes: A distributed waf file system. In Dahlia Malkhi and Dan Tsafir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 135–148. USENIX Association, 2019.
- [33] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. Alleviating garbage collection interference through spatial separation in all flash arrays. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 799–812, 2019.
- [34] Youngjae Kim, Junghee Lee, Sarp Oral, David A Dillow, Feiyi Wang, and Galen M Shipman. Coordinating garbage collection for arrays of solid-state drives. *IEEE Transactions on Computers*, 63(4):888–901, 2012.
- [35] Gunjae Koo, Kiran Kumar Matam, Te I. H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: trading communication with computing near storage. In Hillery C. Hunter, Jaime Moreno, Joel S. Emer, and Daniel Sánchez, editors, *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, pages 219–231. ACM, 2017.
- [36] Pradeep Kumar and H. Howie Huang. Falcon: Scaling IO performance in multi-ssd volumes. In Dilma Da Silva and Bryan Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 41–53. USENIX Association, 2017.
- [37] Kyber. multiqueue i/o scheduler. 2017. <https://lwn.net/Articles/720071/>.

- [38] Jing Li, Peng Li, Rebecca J. Stones, Gang Wang, Zhongwei Li, and Xiaoguang Liu. Reliability equations for cloud storage systems with proactive fault tolerance. *IEEE Trans. Dependable Secur. Comput.*, 17(4):782–794, 2020.
- [39] Shaohua Li. raid5: make stripe handling multi-threading. <https://lwn.net/Articles/563142/>.
- [40] Shaohua Li. raid5 offload stripe handle to workqueue. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=851c30c9badfc6b294c98e887624bfff53644ad21>.
- [41] Yongkun Li, Patrick P. C. Lee, and John C. S. Lui. Analysis of reliability dynamics of SSD RAID. *IEEE Trans. Computers*, 65(4):1131–1144, 2016.
- [42] Yongkun Li, Biaobiao Shen, Yubiao Pan, Yinlong Xu, Zhipeng Li, and John C. S. Lui. Workload-aware elastic striping with hot data identification for SSD RAID arrays. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 36(5):815–828, 2017.
- [43] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A multicore-accelerated file system for flash storage. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 877–891, 2021.
- [44] Linux. Linux perf. <https://perf.wiki.kernel.org/>.
- [45] Linux. Linux raid. https://raid.wiki.kernel.org/index.php/Linux_Raid.
- [46] Linux. Hdfs-raid. 2020. <https://wiki.apache.org/confluence/display/HADOOP2>.
- [47] Shuyang Liu, Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. Analysis of and optimization for write-dominated hybrid storage nodes in cloud. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*, pages 403–415. ACM, 2019.
- [48] Stathis Maneas, Kaveh Mahdavian, Tim Emami, and Bianca Schroeder. A study of SSD reliability in large scale enterprise storage deployments. In Sam H. Noh and Brent Welch, editors, *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 137–149. USENIX Association, 2020.
- [49] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. In Bill Lin, Jun (Jim) Xu, Sudipta Sen Gupta, and Devavrat Shah, editors, *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Portland, OR, USA, June 15-19, 2015*, pages 177–190. ACM, 2015.
- [50] Richard R. Muntz and John C. S. Lui. Performance analysis of disk arrays under failure. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 162–173. Morgan Kaufmann, 1990.
- [51] Dushyanth Narayanan, Austin Donnelly, and Antony I. T. Rowstron. Write off-loading: Practical power management for enterprise storage. In Mary Baker and Erik Riedel, editors, *6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA*, pages 253–267. USENIX, 2008.
- [52] Gal Ofri. raid5 avoid device lock in read one chunk. <https://github.com/torvalds/linux/commit/97ae27252f4962d0fcc38ee1d9f913d817a2024e>.
- [53] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.
- [54] Satadru Pan, Theano Stavrinou, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P., Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, J. R. Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook’s tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 217–231, 2021.
- [55] Tirthak Patel, Suren Byna, Glenn K. Lockwood, Nicholas J. Wright, Philip H. Carns, Robert B. Ross, and Devesh Tiwari. Uncovering access, reuse, and sharing characteristics of i/o-intensive files on large-scale production HPC systems. In Sam H. Noh and Brent Welch, editors, *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 91–101. USENIX Association, 2020.
- [56] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988*, pages 109–116, 1988.
- [57] Sundeep Prakash, Yann Hang Lee, and Theodore Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5):548–559, 1994.

- [58] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In Angela Demke Brown and Florentina I. Popovici, editors, *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, pages 67–80. USENIX Association, 2016.
- [59] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott A. Brandt. Flash on rails: Consistent flash performance through redundancy. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 463–474. USENIX Association, 2014.
- [60] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *login Usenix Mag.*, 41(1), 2016.
- [61] Michael Hao Tong, Robert L. Grossman, and Haryadi S. Gunawi. Experiences in managing the performance and reliability of a large-scale genomics cloud platform. In Irina Calciu and Geoff Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 973–988. USENIX Association, 2021.
- [62] Midhul Vuppapapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 449–462. USENIX Association, 2020.
- [63] Jiguang Wan, Wei Wu, Ling Zhan, Qing Yang, Xiaoyang Qu, and Changsheng Xie. Deft-cache: A cost-effective and highly reliable SSD cache for RAID storage. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*, pages 102–111. IEEE Computer Society, 2017.
- [64] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. Graphwalker: An i/o-efficient and resource-friendly graph analytic system for fast and scalable random walks. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 559–571, 2020.
- [65] Wei Wang, Tao Xie, and Abhinav Sharma. SWANS: an interdisk wear-leveling strategy for RAID-0 structured SSD arrays. *ACM Trans. Storage*, 12(3):10:1–10:21, 2016.
- [66] Suzhen Wu, Haijun Li, Bo Mao, Xiaoxi Chen, and Kuan-Ching Li. Overcome the gc-induced performance variability in ssd-based raids with request redirection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):822–833, 2018.
- [67] Suzhen Wu, Bo Mao, Xiaolan Chen, and Hong Jiang. LDM: log disk mirroring with improved performance and reliability for ssd-based disk arrays. *ACM Trans. Storage*, 12(4):22:1–22:21, 2016.
- [68] Suzhen Wu, Weiwei Zhang, Bo Mao, and Hong Jiang. Hotr: Alleviating read/write interference with hot read data replication for flash storage. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, pages 1367–1372, 2019.
- [69] Suzhen Wu, Weidong Zhu, Guixin Liu, Hong Jiang, and Bo Mao. Gc-aware request steering with improved performance and reliability for ssd-based raids. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*, pages 296–305, 2018.
- [70] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. Lessons and actions: What we learned from 10k ssd-related storage system failures. In Dahlia Malkhi and Dan Tsafir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 961–976. USENIX Association, 2019.
- [71] Gaoxiang Xu, Dan Feng, Zhipeng Tan, Xinyan Zhang, Jie Xu, Xi Shu, and Yifeng Zhu. RFPL: A recovery friendly parity logging scheme for reducing small write penalty of SSD RAID. In *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05-08, 2019*, pages 23:1–23:10. ACM, 2019.
- [72] Gaoxiang Xu, Zhipeng Tan, Dan Feng, Yifeng Zhu, Xinyan Zhang, and Jie Xu. Cap: Exploiting data correlations to improve the performance and endurance of SSD RAID. In *36th IEEE International Conference on Computer Design, ICCD 2018, Orlando, FL, USA, October 7-10, 2018*, pages 59–66. IEEE Computer Society, 2018.
- [73] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND ssds. In *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, pages 15–28, 2017.

- [74] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *FAST 2020*.
- [75] Guangyan Zhang, Zican Huang, Xiaosong Ma, Songlin Yang, Zhufan Wang, and Weimin Zheng. RAID+: deterministic and balanced data distribution for large disk enclosures. In *16th USENIX Conference on File and Storage Technologies, FAST 2018, Oakland, CA, USA, February 12-15, 2018*, pages 279–294, 2018.
- [76] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end data integrity for file systems: A ZFS case study. In Randal C. Burns and Kimberly Keeton, editors, *8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010*, pages 29–42. USENIX, 2010.
- [77] Yong Zhao, Kun Suo, Xiaofeng Wu, Jia Rao, Song Wu, and Hai Jin. Preemptive multi-queue fair queuing. In Jon B. Weissman, Ali Raza Butt, and Evgenia Smirni, editors, *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2019, Phoenix, AZ, USA, June 22-29, 2019*, pages 147–158. ACM, 2019.
- [78] You Zhou, Fei Wu, Weizhou Huang, and Changsheng Xie. Livessd: A low-interference RAID scheme for hardware virtualized ssds. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 40(7):1354–1366, 2021.



VINTER: Automatic Non-Volatile Memory Crash Consistency Testing for Full Systems

Samuel Kalbfleisch
Karlsruhe Institute of Technology

Lukas Werling
Karlsruhe Institute of Technology

Frank Bellosa
Karlsruhe Institute of Technology

Abstract

Non-volatile memory (NVM) is a new byte-addressable storage technology that is part of the processor’s memory hierarchy. NVM is often exposed to applications via an in-kernel file system. To prevent data loss in the case of crashes, the file system implementation needs to be crash-consistent. Achieving crash consistency is difficult however, as special primitives need to be inserted at appropriate places in the program to ensure persistency in the presence of volatile caches.

We introduce VINTER, a new approach to automated NVM crash consistency testing designed for full systems, including unmodified kernel software such as file systems. By tracing NVM accesses of a full system via dynamic binary translation, we capture interactions between user and kernel space code. With such traces, our system efficiently generates relevant crash states using a heuristic that determines NVM locations significant for crash consistency. Finally, it extracts the semantic representation of each crash state. This makes the automatic detection of operation-spanning violations of crash consistency properties such as atomicity feasible. Our approach further aids in fixing detected bugs by representing how bugs originate from simulated crashes which are annotated by trace metadata.

Our evaluation on NVM file systems uncovers several previously unknown bugs, including bugs in the state-of-the-art file systems NOVA and NOVA-Fortis that lead to atomicity violations and data loss.

1 Introduction

Non-volatile memory (NVM) is a new storage technology that is byte-addressable and integrated in the processor’s memory system. Applications can obtain a virtual memory mapping to access non-volatile memory pages directly with load and store instructions [40, 41, 52]. Such direct access enables persistency without a serialization step, but requires extensive reworking of the applications to ensure crash consistency: Modifications need to be flushed from volatile caches, and

developers need to employ memory fences to enforce persistency ordering. Thus, programming for NVM has turned out to be difficult in practice [39].

As an alternative, unmodified applications can benefit from high-speed NVM by running on NVM file systems [5, 23, 26, 45, 48–51, 54, 56] which implement common user space interfaces such as POSIX [15]. Internally, these file systems store metadata and file data directly on NVM, which means that the same challenges for achieving crash consistency apply to these file systems as well.

Recent research has produced many approaches to detecting crash consistency bugs [8, 11, 27–30, 33]. However, we find that most of these approaches cannot easily be applied to file systems. Kernel software is often not supported at all or requires extensive code modification. In particular, static code analysis [11] and symbolic execution [33] are difficult to apply to full systems where a variety of user and kernel space code may interact. It is possible in theory to adapt a kernel file system to user space in order to apply a testing tool designed for user space software. However, this approach is likely to distort results, as the interaction between user and kernel space code can be a source of consistency bugs. For example, the NOVA file system [49] relies on memory barriers being performed when returning to user space.

We introduce VINTER, the virtualization-based NVM tester, a novel approach for testing crash consistency that supports full systems. Using binary translation, we trace relevant instructions such as load-store instructions or barriers in a virtual machine running unmodified kernel and user space code. From that trace, we generate *crash images* which represent possible NVM contents after a crash. We reduce the exponential search space by identifying NVM locations where the recovery code reads from the crash image. By extracting the *semantic state* of each crash image (e.g., a listing of all files in a file system), we can finally automatically verify crash consistency properties such as atomicity.

We use our prototype to test the NVM file systems NOVA [49], NOVA-Fortis [50] and PMFS [10] for crash consistency bugs. We find several new bugs in these file systems

ranging from atomicity violations to data loss and broken file system states. One specific bug we find in NOVA highlights the importance of testing unmodified software instead of higher-level methods such as manual code annotation: A generic Linux helper function for an uncached memory copy has an optimized assembly implementation for x86 that leaves unaligned data in the cache. NOVA uses this function to copy file data to NVM and is thus susceptible to data loss.

We identify the following major contributions of our work:

- Our solution traces NVM accesses using full system emulation with dynamic binary translation. It thus supports unmodified user and kernel space software.
- We apply heuristics to achieve efficient exploration of crash states, avoiding combinatorial state explosion.
- Through grouping simulated crashes by their semantic state, we introduce *automatic* testing of operation-spanning crash consistency properties such as atomicity.
- To help developers fix uncovered bugs, we introduce a representation of semantic crash states and their origins in simulated crashes which are annotated by trace metadata.
- Using our solution, we provide the first comprehensive analysis of NVM file systems for crash consistency.

2 Background and Related Work

In this section, we introduce the memory persistency model that VINTER builds upon and the crash consistency properties it can verify automatically. Then, we discuss related work.

2.1 Memory Persistency Models

Applications that access NVM need to pay close attention to the *memory persistency model* which codifies the architecture’s guarantees about persistency. Multiple models have been proposed [5, 12, 18, 35]. We implement VINTER for the x86 architecture [17, 37] whose persistency model is based on *persistency epochs* [20]. The epoch model divides thread execution into persist epochs and guarantees that stores between epochs are strictly ordered, but not within the same epoch [5, 35]. Stores are buffered in x86 and need to be flushed from volatile caches with instructions such as `clwb` to be persisted [22, 37]. Alternatively, *non-temporal stores* bypass the caches. Memory barriers in the form of `fence` instructions provide *ordering points* that divide the epochs. In the following, we refer to all instructions that are relevant for the persistency model as *persistency primitives*.

2.2 NVM Crash Consistency

Crash consistency as a property of stateful applications is often only informally specified. Throughout this work, we use the following definitions and assumptions about the tested applications.

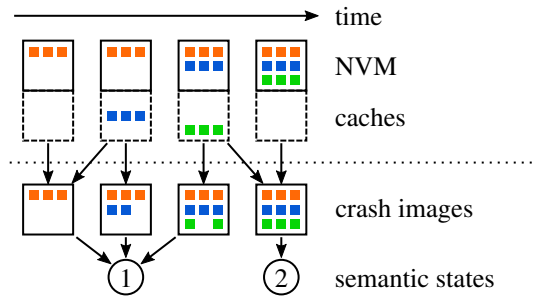


Figure 1: An atomic operation: All crash images (partially shown) recover to one of two semantic states.

We assume that the application keeps its persistent state in NVM. Depending on the access method, modifications may go through a volatile cache or may be reordered. After a crash, these modifications may be partially lost. Consequently, the NVM then contains all fully-persisted data (e.g., through cache flush instructions and memory fences) with a subset of the in-flight modifications applied. We call this a *crash image*.

The period during which a crash may occur is called the *pre-failure* execution [29]. When restarting the application from a crash image, its *post-failure recovery* code will read the data to recover its semantic state. By comparing all possible recovered states that can arise from crashes, we can describe the following crash consistency properties. An operation with all-or-nothing semantics fulfills *atomicity*. More formally, all crash images possible during an atomic operation result in one of two states: either the initial or the final state. Figure 1 illustrates such an atomic operation. From left to right, an application is modifying NVM contents step-by-step. As long as the modifications are not flushed from the caches, crash images with partial cache contents are possible. However, all but the final crash image recover to an identical semantic state. We can observe patterns like this when techniques such as journaling are in use. The final writes (green) mark the previous modifications as valid.

For non-atomic operations, in Section 3.4.1 we define a second, weaker property *Single Final State*: at the end of an operation, all possible crash images recover to the same semantic state. Either of these properties may be violated by different kinds of bugs. For example, a missing memory fence at the end of an operation may violate Single Final State: the CPU could reorder cache flushes, which results in crash images that miss some modifications.

Multiple methods exist that help applications to achieve crash consistency, including logging [46, 47], log structuring [4, 14, 24, 49, 55], and shadow paging [31, 34].

2.3 NVM Crash Consistency Testing

Since making NVM programs crash-consistent is difficult for developers, multiple testing approaches have been proposed

with varying degrees of automation. Many of the previous approaches rely on user-provided code annotations [29, 30], do not automatically confirm bug candidates [8, 29, 30, 33], and are limited to user space applications [11, 29, 33] or would need additional effort like kernel modification [8, 28, 30]. VINTER is free from these limitations.

Lantz et al.'s Yat [27] traces its target in a hardware-assisted hypervisor, constructs all possible crash images by brute force, and tests these by running an integrity checker such as `fscck`. It is able to test kernel space applications, but it relies on code modifications to trap persistency primitives due to limits of hardware-assisted virtualization. Due to its exhaustive crash image exploration, some test cases would take years to complete. Furthermore, Yat can only detect inconsistencies discovered by an integrity checker, and is not able to consider consistency across entire operations.

Liu et al.'s PMTest [30] requires that developers annotate their source code with persistency assertions. Their correctness is then evaluated at runtime. The obvious drawback is that the approach entirely depends on the quantity and quality of annotations which need considerable effort by developers.

XFDetector by Liu et al. [29] applies heuristics to detect a typical bug pattern. During runtime, it looks for read operations during post-failure recovery whose corresponding stores from pre-failure execution have not been explicitly persisted. To avoid false positive bugs, they require manual code annotations. Still, the bug candidates need manual screening.

Neal et al.'s Agamoto [33] applies symbolic execution to NVM user space programs. This allows arguing over many possible execution paths at once and can work with symbolic NVM. It only detects some bug patterns like unpersisted NVM locations after program termination. Thus, false positives can occur and manual vetting is required.

WITCHER by Fu et al. [11] focuses on testing user space key-value stores. The authors propose a testing pipeline that automatically confirms bugs. It uses heuristics based on dynamic and static analysis to choose interesting crash states and thus avoids exhaustive search. Use of static analysis and need for recompilation makes it difficult to apply to full systems, so we propose a new crash image generation heuristic solely based on dynamic analysis. WITCHER automatically detects violations of the *atomicity* of operations [18] by introducing “output equivalence checking” that compares with oracle states obtained before and after an operation. Our work extends upon this by also testing for a relaxation of atomicity. Furthermore, VINTER outputs a representation of all encountered semantic crash states for easy manual sighting when strong properties such as atomicity do not apply. VINTER also passes metadata through its testing pipeline in order to facilitate debugging and root cause analysis of uncovered bugs.

PMFuzz by Liu et al. [28] uses fuzzing for test case generation and is orthogonal to our work. PMDebugger by Di et al. [8] focuses on efficiently processing memory traces.

Formal verification of NVM programs has been proposed [7, 13, 18].

2.4 File System Crash Consistency

The need for file systems to tolerate crashes is not new. Numerous works have proposed methods for checking crash consistency properties in file systems [19, 21, 25, 32, 36, 53]. Most of these approaches rely on instrumentation at the block layer and thus cannot be applied to NVM file systems. However, we see opportunities to adopt specific techniques to an NVM context. CrashMonkey [32] automatically generates test cases for a crash consistency checker. Our evaluation relies on manually written tests, but could be extended with a similar technique. Jaffer et al. [19] evaluate how file systems react to media errors common on solid state drives. Our approach could be extended in this direction, however, there is currently limited information on specific NVM media errors.

File system semantics. A common issue with file system crash consistency checking is that consistency semantics vary from file system to file system and are often only loosely specified. The POSIX standard [15], whose API most Unix file systems implement, does not define crash consistency semantics at all. Bornholt et al. [3] improve on this situation by creating crash consistency models for a few commonly used file systems. Rebello et al. [38] specifically look at error handling by file systems and applications for the `fsync` system call. These issues also apply to NVM file systems. For our analysis, NOVA [49], NOVA-Fortis [50] and PMFS [10] give strong atomicity guarantees for all metadata and file data operations, so complex models were not required.

3 Approach

We introduce a novel crash consistency testing approach to automatically finding bugs that violate crash consistency properties in unmodified NVM applications, including kernel space software. The key requirements for our solution are:

- To support kernel software such as file systems, it should work with *full systems* and should not be limited to user space software.
- It should not require manual development effort. In particular, no code annotations should be needed. Our solution even works with *unmodified* kernel and user space binaries without needing source code access.
- Reported crash consistency bugs should be *automatically confirmed*, and not be based on heuristics that allow for false positives.
- For good performance, it should *avoid exhaustive search* over all possible crash states, and only consider crash states that are likely to exhibit crash consistency bugs.
- It should not only look for single crash states that are obviously broken, but also consider *semantic, operation-spanning crash consistency* such as atomicity. For ex-

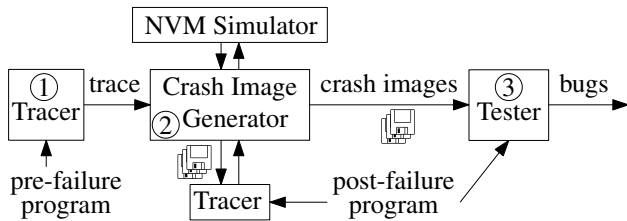


Figure 2: Overview of the testing pipeline. Components are in boxes; arrows are labeled with inputs or produced artifacts.

ample, this allows to determine whether a file system operation is atomic or has invalid intermediate crash states.

3.1 Overview

VINTER dynamically analyzes the execution of a system that interacts with NVM. The goal is to simulate crashes that may occur during a recorded pre-failure execution and find inconsistent states. The proposed framework consists of a testing pipeline made up of multiple components which we depict in Figure 2.

The pipeline’s initial input is an operating system image, which includes all kernel and user space binaries, as well as a sequence of operations that specifies the program’s pre-failure execution during which crashes should be simulated. Each step of the pipeline outputs an artifact that is fed to the next stage of the pipeline. In the final step, uncovered crash consistency bugs and a representation of encountered semantic crash states are output. The entire pipeline may be run multiple times with different test cases. Achieving comprehensive results depends on the quality and quantity of these cases being run in the system. Finding these is orthogonal to our work (e.g., PMFuzz [28]).

VINTER’s *Tracer* makes use of full system emulation with dynamic binary translation to trace NVM operations performed by a system (§3.2). The resulting trace is fed into our *Crash Image Generator*. To be able to reconstruct arbitrary crash states, the Crash Image Generator uses an *NVM simulation* of a given memory persistency model to replay the trace on it (§3.3). At each ordering point (e.g., memory barrier), crash images are chosen by a heuristic that only considers in-flight stores at memory locations likely to be read by the recovery (*post-failure*) code, which avoids exhaustive exploration of crash states. The latter are determined by running the post-failure code on special crash images, hence the Crash Image Generator uses its own Tracer instance.

The *Tester* component finally takes crash images and extracts their *semantic (crash) states* (§3.4). Each of the pre-failure operations is then analyzed for violations of crash consistency properties such as atomicity. For example, if during an operation three different semantic crash states are observed,

the operation has been shown not to be atomic. Uncovered bugs are finally output to the user of the framework, together with a representation of encountered semantic crash states and how they originate from the simulated crashes (§3.5).

The workflow of our testing pipeline is similar to the one in the WITCHER framework proposed by Fu et al. [11]. However, our crash image generation algorithm additionally makes use of the Tracer as part of our heuristic for choosing crash images (we further compare our solution in §2.3).

3.2 Tracer

The Tracer is a full system emulator and is used in the testing pipeline for the following purposes:

- During pre-failure execution, we trace *writes* to NVM as well as invocation of *persistency primitives*, as this allows reconstruction of arbitrary NVM crash states (§3.3).
- Further, we trace *reads* to NVM during post-failure executions as part of our crash image generation heuristic described in Section 3.3.
- Optionally, the Tracer may be used by the Tester component for running the post-failure recovery on the generated crash images to extract their semantic states (§3.4). Tracing NVM accesses may be disabled, but hypercalls can be used (see below).

In common architectures, NVM is accessed directly via load and store instructions to virtual memory that is backed by NVM. Further, persistency primitives (such as cache line flushes and memory fences) are usually implemented as dedicated instructions. To intercept these events, we base the Tracer’s processor emulation on *dynamic binary translation* [43]. This allows full control over the emulated system, and is transparent to the tested program stack, thus binaries may remain unmodified. In contrast, hardware-assisted virtualization (as used by Yat [27]) is faster, but does usually not provide hooks for all events that are of interest (Yat requires recompilation).

During the translation of basic blocks from the emulated architecture to the host architecture, the Tracer adds instrumentation code for persistency primitives to the translated code. Further, the memory abstraction layer of the emulator intercepts memory writes and reads to address ranges backed by simulated NVM. The intercepted events are then output in a *trace log* with associated data. Data includes instruction operands (e.g., cache flush addresses) or memory contents updated by a store to NVM.

Hypercalls. We allow the emulated system to signal the following events as hypercalls to the testing framework, which are additionally recorded in the trace log:

- *Checkpoint* hypercalls during pre-failure execution separate semantic operations. We remind that input to the testing framework is a sequence of operations that will each be separately tested for operation-spanning crash consistency. In practice, this may be in form of a sim-

ple user space executable that calls the tested program's operations and emits hypercalls in between.

- *Success indication* hypercalls may occur during post-failure recovery. If none is emitted, it is assumed that recovery from the specific crash image has failed.

The use of hypercalls is not mandatory for the approach, but it improves the testing workflow for example by allowing separation of operations during pre-failure execution.

Metadata. During pre-failure execution, we log metadata about traced events (such as NVM stores), particularly their location in form of a call stack as traced by the emulator. Although not required to detect crash consistency bugs, it helps a developer to investigate uncovered crash consistency bugs. We retain this metadata when generating crash images later in the pipeline as a way to understand how a crash needs to occur to lead to a certain invalid semantic crash state. Further, additional events helpful for debugging may be traced and logged, such as system calls.

3.3 Crash Image Generator

The Crash Image Generator component takes a pre-failure trace as its input, and it outputs crash images to be passed into the Tester (§3.4). A crash image is a string that describes the binary contents of the NVM in a single crash state that can occur according to the memory persistency model at an arbitrary point of program execution.

Motivation and challenges. In memory persistency models that are based on persistency epochs [5, 35] (including x86 [20]), the set of all possible crash images can be constructed as follows. At each ordering point (usually memory fences), apply any possible subset of potentially unpersisted in-flight stores on top of the memory contents that are already guaranteed to be persisted [8, 11, 27–30, 33]. This works because stores can only become irreversibly persistent at ordering points. Not all subsets may be allowed and order of stores can be relevant depending on the specific memory model [35, 37, 42], such as x86 with intra-cache-line ordering.

Although considering any possible crash state would be comprehensive, it is impractical. Yat by Lantz et al. [27] does so, but the authors find that some test cases would take several years to complete due to exponential explosion in number of crash images. Thus, many other approaches do not actually generate and test crash images, but only apply heuristics on the observed program execution to detect typical bug patterns [8, 29, 30, 33]—this either allows false positives to occur or requires extensive manual code annotations.

Solving this problem requires reducing the number of tested crash images. The chosen subset of crash images should ideally not hide bugs. Accordingly, the chosen crash images should have some properties that makes them relatively likely to exhibit bugs. The recent WITCHER framework proposed by Fu et al. [11] aims to solve this challenge for user space key-value stores: From a mix of static and dynamic analysis, they

infer *likely invariants* regarding the persistency of program data that have presumably been intended by the programmer. Then, they choose crash images that violate these invariants and test if these indeed violate crash consistency. This approach works well for key-value stores, but particularly the reliance on static analysis makes it difficult to apply to full systems, where a variety of user and kernel space code may interact.

Proposed crash image generation heuristic. In contrast, our heuristic only relies on dynamic analysis. For determining crash consistency, only the semantic state as recovered by the application in post-failure recovery is relevant (§3.4). Conversely, the semantic crash state can only have been influenced by *memory locations from which the post-failure execution has read*. Our crash image generation heuristic is based on this idea. When choosing subsets of in-flight stores during crash image generation, our heuristic only considers stores likely to be read by the post-failure recovery. Other stores may be ignored.

We observe that techniques such as journaling and log structuring, both common in file systems, cause the following access pattern: The program writes a journal entry, flushes it completely to NVM and only marks it valid after a store fence. The journal entry may have an arbitrary size, resulting in a large number of in-flight stores. However, considering subsets of these stores for crash image generation is not useful. After a failure, the recovery would only read a journal entry that is marked valid.

According to this observation, we base the decision on whether a store is likely to be read in post-failure recovery on the following assumption: If an NVM location is not read during the post-failure stage on the image where all unpersisted stores are applied, the post-failure stage will likely also not read this location when an arbitrary subset of in-flight stores is applied. Therefore, the heuristic limits the generated crash images to variations of stores that *are* likely to be read during the post-failure stage's execution. As with any heuristic, the assumption may not always hold. We evaluate its effectiveness in Section 6.2.

For the heuristic to capture all relevant NVM locations, the post-failure recovery should read all relevant state, for example by running code that serializes all state (as in the state extractor to be introduced in Section 3.4).

The heuristic's underlying idea of observing interactions between pre- and post-failure executions is based on Liu et al.'s XFDetector framework [29]. However, XFDetector uses these observations directly to detect bug patterns, but does not automatically confirm bug candidates. It further requires developers to manually annotate their code to mark memory belonging to commit variables; it is further not directly compatible with checksumming mechanisms as used by file systems.

NVM simulation. To be able to replay the trace and reconstruct the possible NVM crash states, we assume a simu-

lator of the architecture’s memory persistency model. It holds the *guaranteed persisted memory* content as a binary string, and further a list of *in-flight stores* that have not yet been explicitly persisted. Each of these stores further retains associated metadata from the trace. It further needs to provide functions for applying stores and persistency primitives (e.g., cache line flushes and ordering points). An efficient data structure and algorithm for processing in-flight stores have been proposed by Di et al. [8].

Resulting algorithm. The algorithm processes the pre-failure trace from beginning to end and replays each NVM write operation and persistency primitive invocation on the NVM simulation. At each ordering point, it generates crash images in the following way:

1. Obtain a copy of the current NVM’s guaranteed persisted memory, and apply all stores on it. We obtain NVM_{full} .
2. Instantiate the Tracer with NVM_{full} as initial NVM contents. Execute and trace the post-failure recovery.
3. Look for “read” operations in that post-failure trace to NVM addresses that have overlapping in-flight stores.
4. Consider all subsets of these cross-failure read in-flight stores, and apply each subset to the guaranteed persisted memory and emit the resulting crash images.
5. Continue with replaying the pre-failure trace.

As an optimization, we ignore ordering points if no stores to NVM have occurred before the last one. Further, if there would be too many subsets of cross-failure read in-flight stores according to a configurable threshold, we choose a random selection of these subsets.

Metadata. It is possible for the same crash image (merely a binary string) to be emitted multiple times, but at different ordering points and with different subsets of stores applied. We deduplicate crash images and attach metadata to each image that describes all its *origins*. Each origin includes the (un)persisted in-flight stores’ metadata (containing the stack trace that led to a store), the last ordering point’s ID, and the last checkpoint’s ID. Crash images are further grouped by checkpoint (i.e., the semantic operation they occur during).

3.4 Tester

In the previous step of VINTER’s testing pipeline, we have generated crash images that simulate crashes at multiple points during the traced pre-failure program execution. In this final step, the Tester component analyzes each operation’s crash images for the occurrence of crash consistency bugs. We begin by defining crash consistency properties (§3.4.1) and then describe how the Tester component discovers violations of the properties from a set of crash images (§3.4.2). Our testing approach is an extension of Fu et al.’s “output equivalence checking” [11].

3.4.1 Crash Consistency Definitions

The central idea is that an NVM image can be mapped to an application-specific well-defined *semantic state* that describes the intended meaning of the persisted data. The same semantic state can possibly be encoded by different NVM images. We use \mathcal{S} to describe the set of semantic states, and \perp ($\perp \notin \mathcal{S}$) to denote that an NVM image is not recoverable from because it is faulty. The *state extractor* function E maps NVM images to semantic states, for which we use the notation $E : \{0, 1\}^* \rightarrow \mathcal{S} \cup \{\perp\}$.

We recall that we allow traces to be separated by *checkpoints* (c_1, c_2, \dots) that are signaled by the running program through hypercalls (§3.2). We define *checkpoint intervals* $[c_i, c_{i+1}]$ —called *operations* in the following—that each induce a subsequence of a trace that contains all its recorded events between checkpoints c_i and c_{i+1} . We use checkpoints to separate different semantic operations during the pre-failure execution traced in the beginning of our testing pipeline. We define two crash consistency properties which, depending on the operation, may be considered a requirement for crash consistency of the operation.

We propose the following new property:

Definition 1 (Single Final State, SFS). A checkpoint c_k is single-final-state crash-consistent ($SFS(c_k)$) if and only if all crash images $N_k \subset \{0, 1\}^*$ that can result from crashes in the trace exactly at checkpoint c_k result in the same state $\neq \perp$, or formally: $\exists s \in \mathcal{S} : \{E(n) \mid n \in N_k\} = \{s\}$.

For $s \in \mathcal{S}$, we write $SFS(c_k, s)$ if c_k is single-final-state crash-consistent and $\{E(n) \mid n \in N_k\} = \{s\}$. Further, an *operation* $[c_i, c_{i+1}]$ is single-final-state crash-consistent if and only if c_{i+1} is single-final-state crash-consistent.

SFS is a property that is useful to require even when an operation is not considered atomic; in that case, intermediate states are allowed, but as soon as an operation returns, a crash may not yield any intermediate states anymore.

We further define the well-known atomicity property in our context:

Definition 2 (Atomicity). An operation $[c_i, c_{i+1}]$ is atomic if and only if c_i and c_{i+1} are both single-final-state crash-consistent, and all crash images $N_{[i, i+1]} \subset \{0, 1\}^*$ that can result from crashes anywhere between checkpoints c_i and c_{i+1} result in either of two states $\neq \perp$, or formally:

$$\exists s_{\text{before}}, s_{\text{after}} \in \mathcal{S} : SFS(c_i, s_{\text{before}}) \wedge SFS(c_{i+1}, s_{\text{after}}) \\ \wedge \{E(n) \mid n \in N_{[i, i+1]}\} = \{s_{\text{before}}, s_{\text{after}}\}.$$

Atomicity means that operations execute, from the point of view after crash recovery, in an all-or-nothing fashion: Either an operation is fully run (s_{after}) or not at all (s_{before}). No intermediate states should occur and all states should be recoverable. This means that no more than two states may be observable after recovering from arbitrary crashes between

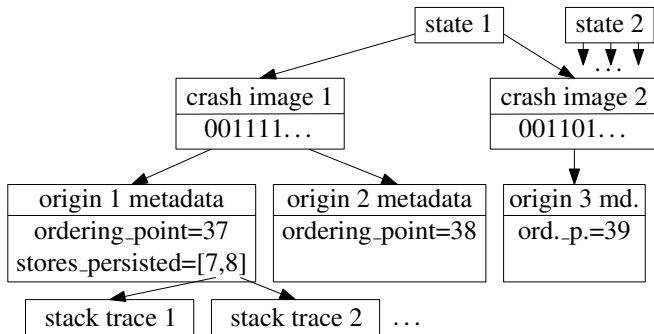


Figure 3: Representation of an operation’s semantic states and their origins’ metadata as output by VINTER.

the two checkpoints of an operation. An atomic operation is obviously also SFS crash-consistent, but SFS crash-consistent operations exist that are not atomic.

The state extractor E will usually be implemented by running the application’s post-failure recovery on the given crash image including a step that serializes the state. Additional tests may be run after the recovery to ensure correctness (such as `fsck` on file systems). In this case, if a hypercall indicating success is not issued by the post-failure recovery, the state is mapped to \perp . The Tracer component can be reused to provide a virtual environment that implements these hypercalls.

3.4.2 Testing Crash Consistency Properties

VINTER’s Tester component finds crash consistency violations given a set of crash images from the previous step of the testing pipeline. As its input, we assume a map of (already deduplicated) crash images to lists of the metadata that describe an image’s (possibly multiple) origins (§3.3).

The Tester builds a table $M_{cp \rightarrow states}$ that maps checkpoint IDs to semantic states which originated from crash images where the most recent checkpoint hypercall had the corresponding ID. As an image can have multiple origins with different checkpoint IDs, a semantic state may be mapped to more than one checkpoint. It also builds another similar table $M_{cp \rightarrow fin. states}$, with the exception that only images are considered that result from crashing directly at a checkpoint boundary, thus only considering the “final” states at the end of an operation.

It is easy to find violations of crash consistency properties by scanning the tables:

- The *Single Final State* property of a checkpoint is valid if and only if $M_{cp \rightarrow fin. states}$ contains exactly one state for that checkpoint.
- The *Atomicity* property of an operation $[c_i, c_{i+1}]$ is valid if and only if c_i and c_{i+1} are SFS crash-consistent (with $M_{cp \rightarrow fin. states}[c_i] = \{s_{before}\}$ and $M_{cp \rightarrow fin. states}[c_{i+1}] = \{s_{after}\}$) and $M_{cp \rightarrow states}[c_i] = \{s_{before}, s_{after}\}$.

Violations of these properties are reported by the Tester.

Further, for each operation it outputs a representation of all encountered semantic crash states with their associated crash images and the crash images’ origins (Figure 3). This representation helps with root cause identification as we cover in the following section.

3.5 Bug Analysis

The Tester’s output lets developers determine whether crash consistency bugs have been uncovered. Trivially, if the extracted semantic states include an unrecoverable state (\perp), a bug has been uncovered. Otherwise, in the simple case, a developer specifies that a certain operation should hold one of the previously covered crash consistency properties, and then a violation reported by the Tester implies a bug. However, the crash consistency semantics of an operation may be more complex or even unclear and may allow multiple intermediate states. The developer can then manually inspect the encountered semantic states for their validity (as output in the maps $M_{cp \rightarrow states}$ and $M_{cp \rightarrow fin. states}$). Alternatively, they can automate the decision by implementing an operation-specific predicate that validates whether for a set of semantic states the operation is crash-consistent.

Once bugs have been discovered, VINTER aids developers in understanding their root causes by representing how crash-consistency-violating semantic states originate from simulated crashes. As depicted in Figure 3, every semantic state keeps a link back to the crash images it was extracted from, and each crash image keeps a link to the simulated crashes’ states (origins) it was created from. As a crash image only represents the NVM’s binary contents, multiple different crash images may lead to the same semantic state (e.g., an uncommitted journal entry does not contribute to the semantic state). Further, crashing at different ordering points or with different subsets of in-flight stores applied, as stored in the “origin metadata,” may lead to the same crash image.

Each crash image origin includes a list of in-flight stores at the ordering point (fence) where the crash was simulated, as well as the subset of in-flight stores applied to obtain the crash image. Each of the ordering points and their in-flight stores is annotated with its stack trace (as logged by the Tracer) and thus maps back to its source code location.

By inspecting the origins of a crash-consistency-violating semantic state, developers can understand where and how a crash needs to occur to trigger the bug (as in source code location and persistency status of NVM data). This eases root cause identification. For example, if *all* of an invalid state’s origins have a *strict subset* of in-flight stores applied (i.e., the crash image at the same ordering point with all in-flight stores applied does not lead to the invalid state), then introducing more persistency ordering constraints (fences) may likely fix the bug.

We present two concrete bug analyses in Section 5.3.

4 Implementation

We implement a prototype of VINTER for the 64-bit x86 architecture [17]. Its tracer is based on PANDA [9] and Py-PANDA [6], which provide a platform for dynamic analysis built on QEMU [1]. QEMU is a full system emulator based on dynamic binary translation. Core parts of our prototype are written in Python. Therefore, while its performance is sufficient for our file system evaluation, considerable improvements are likely possible without changing the general approach. VINTER’s prototype is available and described in [Appendix A](#).

Our NVM simulator implements x86’s memory persistency model [17, 37]. By dividing the simulated NVM into segments of 64 bytes and keeping ordering of in-flight stores within these lines, VINTER respects intra-cache-line ordering constraints. This guarantees ordering even of non-temporal stores, which is a higher guarantee than the architecture gives. Setting the segment size to 8 bytes would allow testing reordering of non-temporal stores, but would also break intra-cache-line ordering which some NVM file systems rely on [10].

5 File System Crash Consistency

The crash consistency testing tool we have described so far does not have any parts specific to file systems and could be applied to arbitrary software running in a virtual machine. In this section, we describe how to apply it to NVM file systems. Then, we present the results of our analysis of NOVA [49], NOVA-Fortis [50] and PMFS [10].

Our testing pipeline requires an application-specific *state extraction procedure* for mapping from a crash image to its semantic state. The Tester component boots a virtual machine with the crash image, runs the procedure, and records the output or a failure state \perp in case of errors. We implement file system state extraction as follows:

1. Mount the file system read-only.
2. Traverse the file system and output a serialized representation of each file.

We need to mount the file system read-only to prevent inadvertent changes to metadata such as file access timestamps. The file systems we target adhere to the POSIX standard [15], which allows us to build a generic state extractor for all POSIX-compatible file systems. For each file (including regular files, directories, symbolic links) we output a serialization of its path, its contents, its type, and most metadata from the *stat* structure¹.

After the state extraction completes, we verify that the file system can still be modified. We remount the file system as writable and run an additional test-case-specific command that modifies some of the files or directories used in the test.

¹We exclude runtime properties such as device IDs and preferred I/O block size.

If any of these operations fail, we mark the state described by this crash image as failure state \perp .

Our notion of consistency only encompasses the file system state as visible via the file system API. The underlying assumption is that if the file system is still properly readable and writable, its state can be considered consistent without the need to inspect its internal state.

For testing more traditional file systems that depend on separate integrity checkers (`fsck`), these could also be run as part of the state extraction procedure and map to the failure state \perp if the integrity checker fails.

5.1 File System Setup

For each tested file system, we need a corresponding virtual machine image. To reduce the time required for tracing, we hand-craft minimal VM images consisting of a statically-linked Linux kernel image with a user space based on Busy-Box [44]. We do not use an init system. Instead, we let the system spawn a shell that accepts test commands over a virtual serial console. Consequently, there are no unrelated processes running in the background during tracing.

5.2 Test Cases and Results

We manually craft 16 test cases consisting of operation sequences which cover most basic file system operations. [Figure 4](#) shows a summary of the test cases and the results. Most test cases correspond to basic file system operations given in monospace font. The “atime” and “[cm]time” test cases update the corresponding timestamps as side effects of a file read or directory operations², whereas “touch” uses a system call for that purpose. We test three variants of the `rename` operation: a rename that overwrites an existing file (`overwrite`), moving a directory into another (`directory`), and changing the file name to a longer one (`long name`). The “long name” test case creates a file with a long file name and writes to it. Test cases with “long” file names use a name that exceeds the cache line size (which is 64 bytes). Finally, “update” modifies a small part in the middle of a larger file.

In total, VINTER finds previously unreported bugs in 7 out of 16 test cases for NOVA. We analyze these bugs manually and find three root causes. First, we observe missing cache flushes when NOVA writes unaligned data to NVM which lead to data loss. This issue manifests in our test cases “write” and “symlink,” but could also occur in “append” and “update” (marked with an asterisk) depending on the length of the written data. The test cases with long filenames (i.e., longer than cache line size) suffer from the same issue and result in files where reading the metadata with `stat` fails. Second, the rename operation is not atomic. We observe crash states where the renamed file or directory is completely missing.

²Adding or removing files from a directory updates the directory’s change and modification timestamps.

	write	append	atime	[cm]time	chmod	chown	link	symlink
NOVA	🔴🔴	✓*	✓	✓	✓	✓	🔴	🔴🔴
NOVA-Fortis	🔴	🔴	✓	🔴	✓	✓	🔴	🔴🔴
PMFS	✓	✓	✓	🔴	✓	✓	✓	✓
	mkdir rmdir	rename overwrite	rename directory	rename long name	touch	long name	unlink	update
NOVA	✓	🔴🔴	🔴🔴	🔴🔴🔴	✓	🔴	✓	✓*
NOVA-Fortis	🔴	🔴🔴🔴	🔴🔴🔴	🔴🔴🔴	🔴	🔴	🔴	🔴
PMFS	🔴🔴	🔴🔴	✓	✓	✓	🔴	🔴🔴	✓

🔴 data loss 🔴 crash 🟡 atomicity violation 🔴 read/write fails after recovery 🔴 multiple final states (SFS violation)

Figure 4: Crash consistency bugs discovered by VINTER.

In the following section, we give a detailed analysis of these two bugs. Third, creating hard links is not completely atomic. Our tool detects a crash state where the original file’s link count (`st_nlink`) is incremented, but the new link does not yet exist.

As NOVA-Fortis is an extension of NOVA with the addition of checksumming and parity, it shares most of the issues we find in NOVA. However, we see three additional failing test cases. Our tool discovers intermediate crash states where both data and the checksum over that data are only partially persisted. This leads to checksum errors during recovery and errors when trying to read or write the affected files. We also observe an instance where the additional data integrity mechanisms help: In the “write” test, NOVA-Fortis does not suffer from data loss since it can recover the unpersisted data from parity. However, it appears that NOVA-Fortis does not protect all data that way: The data loss in the “symlink” test case shares the same root cause as for “write,” but still occurs in NOVA-Fortis.

PMFS suffers from fewer failing test cases than the NOVA variants. We observe a minor atomicity violation in test cases that remove or overwrite a file. Before the file disappears, crash states exist where the file has updated change and modification timestamps. A more serious issue can occur when removing files in the root directory: Crash states are possible where mounting the file system results in a failing assertion, which leads to a crash of the PMFS kernel module.

We reported all NOVA and NOVA-Fortis bugs to the developers³. We did not report PMFS issues since it is not maintained anymore.

5.3 Analysis

VINTER has discovered several previously unreported crash consistency bugs in the NOVA variants. In the following, we exemplarily provide a detailed analysis of two of these bugs that highlights advantages of our crash consistency testing

³Issue IDs 105, 116, 121–125; each accessible at <https://github.com/NVSL/linux-nova/issues/<ID>> (also on archive.org)

unpersisted stores	call stack of store (metadata)
0x2db008 ↦ 1	↳ <code>__copy_user_nocache</code>
0x2db009 ↦ d	↳ <code>do_nova_inplace_file_write</code>
	↳ ...
0x2db00a ↦ ↓	↳ <code>vfs_write</code>
	↳ ...

Figure 5: The unpersisted stores after writing `HelloWorld↓` to a file in NOVA. The bytes 1, d, ↓ are the last three bytes of the string.

event	operand	instr. (metadata)
syscall	<code>sys_write(fd=1, buf='HelloWorld\n', n=11)</code>	
...
write (NT)	<code>0x...0 ↦ HelloWor</code>	<code>movnti qword [rdi], r8</code>
write (T)	<code>0x...8 ↦ 1</code>	<code>mov byte [rdi], al</code>
write (T)	<code>0x...9 ↦ d</code>	<code>mov byte [rdi], al</code>
write (T)	<code>0x...a ↦ ↓</code>	<code>mov byte [rdi], al</code>
fence		<code>sfence (store fence)</code>

Figure 6: Excerpt of the trace that shows how the file contents are written to NVM. No flush operations follow on the cache line belonging to the temporal stores.

approach. Furthermore, the analysis illustrates how VINTER helps manual analysis with the metadata carried throughout the testing pipeline and presented as part of the Tester’s report.

5.3.1 Incompletely Persisted Data

We first analyze the data loss bug occurring in the test case “write” (see Figure 4). The test case creates a file and writes the string `HelloWorld↓`, as in the shell command `echo HelloWorld > /mnt/myfile`. Our tool detects a violation of Single Final State and records partially persisted file contents where up to three bytes at the end are replaced with zero (e.g., `HelloWor000` and `HelloWorl000`).

First, we take a look at the unpersisted stores as reported

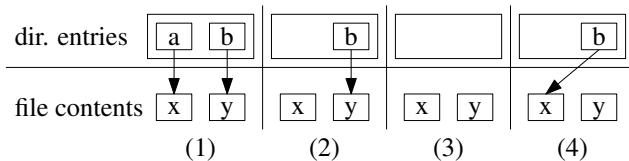


Figure 7: Observed crash states in NOVA during a rename of file *a* to *b* in a directory, intending to replace *b*. Boxes represent files. (1) is the initial state.

by the Tester component. As pictured in Figure 5, each store is associated with a call stack. We see that the stores originate from a `write` system call and that NOVA uses the function `__copy_user_nocache` to write to NVM.

Next, we inspect the NVM trace pictured in Figure 6. It captures the system call entry as well as all writes to NVM, cache flushes, and memory fences. Non-temporal writes (NT) are distinguished from temporal writes (T) as the former bypass the volatile caches. We see that the helper function `__copy_user_nocache` writes the first eight bytes of the string to NVM with a single non-temporal store, then writes the remaining bytes one by one with temporal stores. As there is no cache flush operation, these bytes end up unpersisted.

Further investigation in the Linux source code shows that `__copy_user_nocache` has an architecture-specific implementation for x86 implemented in assembly. As the function was written with performance and not persistency in mind, the Linux developers deemed it acceptable to use temporal stores for unaligned data.

This bug highlights the importance of testing *unmodified* software. Using an approach that relies on source code annotations to trace NVM events, a developer would likely assume that the Linux helper function works correctly and would annotate it accordingly without following through to the architecture-specific assembly implementation.

5.3.2 Data Loss During Rename

Second, we analyze the atomicity issues of the `rename` operation in NOVA. As depicted in Figure 4, our tool detects atomicity violations and data loss for all variants we test (overwriting a file, renaming a directory, and a normal rename with long file names). We visualize the crash states for the overwrite variant in Figure 7. In addition to the initial state (1) and the desired final state (4), we observe a state where the file to be renamed is missing (2), as well as a state where the target file is also missing (3). For the other two test cases that do not overwrite files, we observe crash states similar to (2) where the file or directory is missing.

We find that both invalid crash states from Figure 7 have at least one crash image associated with an “origin” where all in-flight stores were persisted (i.e., includes all volatile cache lines; §3.5). Consequently, the bug does not stem from mistakes with NVM persistency primitives. We inspect the call

stacks for these crash images and find that they correspond to different locations in the `nova_rename` function. By inserting early `return` statements at these locations, we can reproduce the issue without our testing framework, thus confirming the issue.

This bug shows that our testing pipeline is also capable of finding persistency issues that are not specific to NVM. By carrying metadata such as stack traces through the pipeline, our framework makes bug verification easy.

6 Evaluation

In the previous section, we have shown that our approach is capable of finding new crash consistency bugs in file systems. We now give a more complete picture of our prototype’s performance by answering the following questions: Is the approach comprehensive or does it miss certain types of known crash consistency bugs? How effective is our crash image generation heuristic at reducing the search space of possible crash images? How big is the slowdown incurred by tracing and how fast are the other stages of the testing pipeline?

6.1 Completeness

It is well known that—outside of formal methods—software testing can only prove the existence of bugs, but not their absence. Nevertheless, we would like to evaluate the *completeness* of our approach: Does it miss certain types of real-world crash consistency bugs?

To this end, we sight all 45 issues and patches available on the public NOVA bug tracker which were submitted between 2019-01-01 and 2021-10-30. We include issues and patches that are related to crash consistency bugs. We exclude #110⁴ as according to the bug description, recovery from a crash only leads to error messages but no observed wrong behavior. We exclude #98, as according to the bug report, it requires an NVM capacity > 128GiB, which is infeasible with our evaluation setup. We are left with four different patches (#89, #92, #95, #109) that fix crash consistency bugs in NOVA and NOVA-Fortis.

To assess whether our prototype is able to find these bugs, we evaluate our prototype on a commit before and after each bug, and compare the Tester component’s reports. This way, we can ignore the crash consistency bugs we describe in Section 5 which may already be present in older NOVA versions.

Our prototype is able to find all of the known bugs chosen by our evaluation methodology. There is no need for special test cases. All bugs are triggered by at least one of the test cases we describe in Section 5.2.

⁴The numbers are IDs of issues or patches in the NOVA bug tracker and can be accessed at URLs of the following form: <https://github.com/NVSL/linux-nova/issues/<ID>> (also on archive.org)

	$\mu \pm \sigma$ [s]
total elapsed Tracer process time	6.37 ± 0.11
↳ boot (only minimal instrumentation)	1.63 ± 0.03
↳ trace (from outside)	4.15 ± 0.10
↳ trace (in guest, portion of command)	3.02 ± 0.05
execution in guest with raw PANDA	0.09 ± 0.00

Figure 8: Runtimes measured while tracing the pre-failure command of the “write” test case in NOVA. 20 runs, μ is mean and σ is sample standard deviation.

6.2 Effectiveness of Heuristic

We evaluate whether our proposed crash image generation heuristic based on cross-failure reads (§3.3) helps with efficiency as intended. To this end, we modify our prototype to consider *all* lines with unpersisted stores for crash image generation, rather than only lines overlapping with the unpersisted cross-failure reads as reported by our heuristic. We run this modified prototype on all NOVA test cases (§5.2).

We compare both the number of unique crash images generated (accumulated over all test cases), and the semantic crash states discovered by the modified prototype with that of the original prototype. 2466 unique crash images are produced by the modified prototype without the heuristic, versus only 438 crash images by the original prototype⁵. Thus, the prototype without the heuristic needs to test approximately 5.6 times as many crash images. This results in an increased runtime, but it still discovers only the same semantic crash states as the original prototype using the heuristic.

We additionally analyze how the heuristic reduces the number of in-flight stores considered for crash image generation. In 47 out of 178 applications of the heuristic during all NOVA tests, the recovery code does not read any in-flight stores. We manually verify that these cases occur during journaling in NOVA by checking tracing metadata. In 94 applications of the heuristic, all cache lines with in-flight stores are read. In the remaining 37 heuristic applications, the cross-failure reads make up a strict subset of in-flight stores.

6.3 Performance

Even though performance was not a priority of our prototype, we evaluate its performance to show that the approach is sufficiently fast for testing file systems. We benchmark the Tracer’s performance with the pre-failure command of the “write” test case (see §5.2) on the NOVA evaluation target on an Intel Xeon E5-2620 v4 CPU. We depict the results in Figure 8. Compared to the runtime in raw PANDA (i.e., binary translation without tracing), we observe a slowdown of approximately factor 34. The resulting traces each have

⁵We have only performed a single run for each test case; the generated crash images in each run can slightly vary due to nondeterministic guest execution.

	$\mu \pm \sigma$ [s]
total elapsed process time	83.82 ± 0.53
↳ boot	1.63 ± 0.03
↳ Crash Image Generator	37.87 ± 0.33
↳ cross-failure tracing (heuristic)	$2.00 \pm 0.28 \times 12$
↳ Tester	43.65 ± 0.35
↳ reset to snapshot & load image	$0.08 \pm 0.01 \times 31$
↳ run dumper command (PANDA)	$1.04 \pm 0.12 \times 31$

Figure 9: Runtimes of the Crash Image Generator & Tester process when processing the trace from Figure 8.

≈ 304438 events and are each ≈ 11.73 MiB in size (in a simple textual format; compressed only ≈ 0.15 MiB).

As the Crash Image Generator and Tester run in the same process in our prototype, we show the execution time of both combined in Figure 9. 12 crash images at fences are used as input for the cross-failure heuristic. The tester processes 31 unique crash images that stem from 77 origins (§3.3) and result in seven unique semantic crash states. We find that the Crash Image Generator and Tester contribute roughly equally to the execution time.

With metadata tracing enabled, the runtime of the Tracer increases significantly to $78.70s \pm 0.75s$, whereas the runtime of the Crash Image Generator and Tester only increases slightly to $84.84s \pm 0.60s$. We argue that performance of metadata tracing mode is not very relevant in practice: Test cases can be tested without metadata tracing, and if a crash consistency bug is uncovered, the affected test cases can be re-run with metadata tracing enabled. Nevertheless, performance can presumably be significantly improved.

In total, all our test cases from Section 5.2 take approximately 24 minutes to execute sequentially on the NOVA evaluation target (without metadata tracing). This includes additional steps such as compressing traces. As test cases can be analyzed in parallel, the whole testing time can be reduced to only a few minutes.

7 Discussion

VINTER fares well at finding new bugs. Its testing pipeline is highly automated and the manual effort required for setup and interpretation of its results is low. In the context of file system testing, the Single Final State property has turned out to be useful, as even if operations are not considered atomic, this property should hold for most file system operations after a call to `sync`. Even in cases where the crash consistency semantics of a file system operation are not clear, VINTER’s representation of semantic crash states makes manual vetting feasible. It has turned out to be useful to test modifying the file system after recovery as part of the state extraction procedure, as some of the bugs only become visible when such an operation fails.

We find that severe crash consistency bugs can be found in NVM file systems by only testing primitive file system operations, with no need for complex interaction between multiple operations. This is contrary to the bugs discovered by CrashMonkey [32], which all appear to involve more complex operation sequences. We see two potential reasons: First, programming for NVM with its byte-level access and persistency semantics is much more complicated than the programming pattern for traditional file systems, where updated sectors are first built in DRAM and then transferred to block storage. Second, the tested file systems are of relatively young age, and are still research prototypes not aimed at production usage (and in case of PMFS, even unmaintained). Testing more complex operation sequences on our evaluation targets might yield even more bugs.

8 Conclusion

Crash consistency is difficult to achieve in non-volatile memory (NVM) software. Existing works on NVM crash consistency testing for kernel software are either inefficient or incomprehensive and not automated, and none consider crash consistency and atomicity among entire operations. This makes existing work unsuitable for comprehensive file system crash consistency testing.

In this work, we have introduced VINTER, a new approach to automatic testing of non-volatile memory software. VINTER consists of an automated testing pipeline that traces executions of full systems that use NVM, simulates crashes, and finally tests the resulting crash images for consistency. We use VINTER to find crash consistency bugs in NVM file systems, including the state-of-the-art file systems NOVA [49] and NOVA-Fortis [50]. Our evaluation uncovers several bugs in all tested file systems, many of them previously unreported. The bugs lead to issues such as data loss, kernel crashes, and unwritable files.

To summarize, our approach is general as it is compatible with different kinds of software including kernel code, easy to apply as it is largely automated and does not need code modifications, and has been shown to find new bugs in existing software.

8.1 Future Work

We lay out possibilities for future work on our subject. We see several areas for improvement:

Support for persistent caches. Some recent processors ensure that all data in the CPU caches is written out to persistent memory in case of a power failure (e.g., eADR [16]). With persistent caches, the programmer no longer needs to use cache flushes to write out data to NVM. This greatly reduces the potential for crash consistency bugs, but does not entirely eliminate them. For example, the NOVA bug we describe in [Section 5.3.2](#)

would still occur on an eADR-enabled system. VINTER could support detecting crash consistency bugs on eADR systems by considering prefixes of all temporal writes instead of arbitrary subsets during crash image generation (§3.3). Choosing subsets of in-flight stores would however still be applicable to non-temporal writes since these are weakly-ordered.

Fault injection. Some NVM software including NOVA-Fortis [50] intends to be resilient against media errors. VINTER could be extended with fault injection to test robustness against corruption.

Heuristics. Bug detection or crash image generation heuristics that observe control or data flow such as those proposed by Fu et al. [11] could be adapted to our approach.

Evaluate NVM operating systems. VINTER could not only be applied to file systems, but also to entire operating systems targeted for NVM, like Bittman et al.'s Twizzler [2]. Twizzler removes the file system from the OS interface, and instead allows applications to directly allocate NVM.

File system test cases. Our file system evaluation could be extended by running automatically generated test cases in a large scale similar to Mohan and Martinez et al.'s CrashMonkey [32].

Traditional file systems. VINTER could also be extended to test traditional block-storage-based file systems without NVM support, bringing features such as automatic atomicity testing over previous work [32]. A generally applicable approach to achieve this would be virtualizing a block storage device and recording a trace of writes as well as flush primitives.

A Artifact Appendix

Abstract

We provide an artifact containing our prototype implementation of VINTER as described in [Section 4](#). The artifact further includes the test cases, configurations, and scripts for reproducing the major parts of our file system analysis (§5) as well as our broader evaluation (§6).

Scope

We aim to achieve two main goals with the artifact. First, it allows reproducing the results of this paper. In particular:

- VINTER can find new bugs in file systems and can help developers with finding the root cause. We provide instructions for reproducing [Figures 4 to 6](#) as well as [Section 5.3](#).
- VINTER can reproduce previously fixed bugs in NOVA. We provide instructions for reproducing [Section 6.1](#).
- VINTER's heuristic is effective at reducing the number of generated crash images without missing semantic states. We provide instructions for reproducing [Section 6.2](#).
- VINTER is sufficiently fast for analyzing file systems. We provide instructions for reproducing [Figures 8 and 9](#).

Second, we provide VINTER for the purpose of analyzing other file systems, in the hope that it will prove useful in developing new NVM file systems.

Contents

Our source code repository (located at `/home/vinter/vinter` in the virtual machine image) contains the following components:

- `README.md` contains general setup information, and `artifact-evaluation/README.md` contains instructions for reproducing the experiments and launching a virtual machine where VINTER is preinstalled.
- `vinter_python/`: The original implementation of VINTER that is used for the analysis in this paper.
 - `pmemtrace.py`: The Tracer component.
 - `trace2img.py`: The Crash Image Generator and Tester components.
 - `trace-and-analyze.sh`: Main script for running the full testing pipeline.
 - `report-results.py`: Script for analyzing output from the testing pipeline.
- `vinter_rust/`: A reimplementaion of VINTER in Rust, with the intention of improved performance and to provide a clean base for future extensions.
 - `vinter_trace/`: The Tracer component.
 - `vinter_trace2img/`: The Crash Image Generator and Tester components. Main entry point for running the full testing pipeline.

- `fs-testing/`: Everything related to the analysis of file systems.
 - `scripts/`: Helper scripts, virtual machine (VM) definitions, and test case definitions.
 - `initramfs/`: BusyBox-based user space of the test VMs.
 - `fs-dump/`: File system state extraction program.
 - `linux/`: Configurations of the Linux kernels we test.
- `panda/`: The underlying full system emulator based on upstream PANDA [9] with patches applied.

Hosting

VINTER's source code is available on GitHub at <https://github.com/KIT-OSGroup/vinter> on the branch `atc22-artifact`, commit `4b7e5651e820ec9ebbe2a7321e28b2748103ab74`.

Additionally, we provide an archive containing a virtual machine image that comes installed with VINTER and all its dependencies at doi:[10.5281/zenodo.6626098](https://doi.org/10.5281/zenodo.6626098). The archive contains instructions for using the virtual machine image.

References

- [1] Fabrice Bellard. “QEMU, a Fast and Portable Dynamic Translator.” In: *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005. URL: <https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator>.
- [2] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. “Twizzler: A Data-Centric OS for Non-Volatile Memory.” In: *ACM Trans. Storage* 17.2 (June 2021). ISSN: 1553-3077. DOI: [10.1145/3454129](https://doi.org/10.1145/3454129).
- [3] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. “Specifying and Checking File System Crash-Consistency Models.” In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’16. Atlanta, Georgia, USA: Association for Computing Machinery, 2016, pp. 83–98. ISBN: 978-1-4503-4091-5. DOI: [10.1145/2872362.2872406](https://doi.org/10.1145/2872362.2872406).
- [4] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. “FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory.” In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 1077–1091. ISBN: 9781450371025. DOI: [10.1145/3373376.3378515](https://doi.org/10.1145/3373376.3378515).
- [5] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. “Better I/O through Byte-Addressable, Persistent Memory.” In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP ’09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 133–146. ISBN: 978-1-60558-752-3. DOI: [10.1145/1629575.1629589](https://doi.org/10.1145/1629575.1629589).
- [6] Luke Craig, Andrew Fasano, Tiemoko Ballo, Tim Leek, Brendan Dolan-Gavitt, and William Robertson. “Py-PANDA: Taming the PANDAmium of Whole System Dynamic Analysis.” In: *Workshop on Binary Analysis Research (BAR)*. Vol. 2021. 2021, p. 21.
- [7] John Derrick, Simon Doherty, Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. “Verifying Correctness of Persistent Concurrent Data Structures.” In: *Formal Methods – The Next 30 Years*. Ed. by Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira. Cham: Springer International Publishing, 2019, pp. 179–195. ISBN: 978-3-030-30942-8.
- [8] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. “Fast, Flexible, and Comprehensive Bug Detection for Persistent Memory Programs.” In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 503–516. ISBN: 978-1-4503-8317-2. DOI: [10.1145/3445814.3446744](https://doi.org/10.1145/3445814.3446744).
- [9] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. “Repeatable Reverse Engineering with PANDA.” In: *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. PPREW-5. Los Angeles, CA, USA: Association for Computing Machinery, 2015. ISBN: 978-1-4503-3642-0. DOI: [10.1145/2843859.2843867](https://doi.org/10.1145/2843859.2843867).
- [10] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. “System Software for Persistent Memory.” In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys ’14. Amsterdam, The Netherlands: Association for Computing Machinery, 2014. ISBN: 978-1-4503-2704-6. DOI: [10.1145/2592798.2592814](https://doi.org/10.1145/2592798.2592814).
- [11] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. “Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores.” In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP ’21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 100–115. ISBN: 9781450387095. DOI: [10.1145/3477132.3483556](https://doi.org/10.1145/3477132.3483556).
- [12] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. “Relaxed Persist Ordering Using Strand Persistency.” In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 652–665. DOI: [10.1109/ISCA45697.2020.00060](https://doi.org/10.1109/ISCA45697.2020.00060).
- [13] Morteza Hoseinzadeh and Steven Swanson. “Corundum: Statically-Enforced Persistent Memory Safety.” In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 429–442. ISBN: 978-1-4503-8317-2. DOI: [10.1145/3445814.3446710](https://doi.org/10.1145/3445814.3446710).
- [14] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. “Log-Structured Non-Volatile Main Memory.” In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 703–717.

ISBN: 978-1-931971-38-6. URL: <https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu>.

- [15] *IEEE Std 1003.1-2017 (revision of IEEE Std 1003.1-2008) – IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(®)) Base Specifications, Issue 7*. IEEE Computer Society and The Open Group.
- [16] Intel. *eADR: New Opportunities for Persistent Memory Applications*. 2021. URL: <https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html>.
- [17] Intel. *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Apr. 2021.
- [18] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. “Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model.” In: *Distributed Computing*. Ed. by Cyril Gavoille and David Ilcinkas. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 313–327. ISBN: 978-3-662-53426-7.
- [19] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. “Evaluating File System Reliability on Solid State Drives.” In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 783–798. ISBN: 978-1-939133-03-8. URL: <https://www.usenix.org/conference/atc19/presentation/jaffer>.
- [20] Jungi Jeong and Changhee Jung. “PMEM-Spec: Persistent Memory Speculation (Strict Persistency Can Trump Relaxed Persistency).” In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 517–529. ISBN: 978-1-4503-8317-2. DOI: [10.1145/3445814.3446698](https://doi.org/10.1145/3445814.3446698).
- [21] Yanyan Jiang, Haicheng Chen, Feng Qin, Chang Xu, Xiaoxing Ma, and Jian Lu. “Crash Consistency Validation Made Easy.” In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 133–143. ISBN: 9781450342186. DOI: [10.1145/2950290.2950327](https://doi.org/10.1145/2950290.2950327).
- [22] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. “Efficient persist barriers for multicores.” In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015, pp. 660–671. DOI: [10.1145/2830772.2830805](https://doi.org/10.1145/2830772.2830805).
- [23] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. “SplitFS: Reducing Software Overhead in File Systems for Persistent Memory.” In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP ’19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 494–508. ISBN: 9781450368735. DOI: [10.1145/3341301.3359631](https://doi.org/10.1145/3341301.3359631).
- [24] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. “SLM-DB: Single-Level Key-Value Store with Persistent Memory.” In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 191–205. ISBN: 978-1-939133-09-0. URL: <https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet>.
- [25] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. “Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework.” In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP ’19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 147–161. ISBN: 9781450368735. DOI: [10.1145/3341301.3359662](https://doi.org/10.1145/3341301.3359662).
- [26] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. “Strata: A Cross Media File System.” In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: Association for Computing Machinery, 2017, pp. 460–477. ISBN: 9781450350853. DOI: [10.1145/3132747.3132770](https://doi.org/10.1145/3132747.3132770).
- [27] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. “Yat: A Validation Framework for Persistent Memory Software.” In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 433–438. ISBN: 978-1-931971-10-2. URL: <https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz>.
- [28] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. “PMFuzz: Test Case Generation for Persistent Memory Programs.” In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 487–502. ISBN: 978-1-4503-8317-2. DOI: [10.1145/3445814.3446691](https://doi.org/10.1145/3445814.3446691).
- [29] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. “Cross-Failure Bug Detection in Persistent Memory Programs.” In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming*

Languages and Operating Systems. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 1187–1202. ISBN: 978-1-4503-7102-5. DOI: [10.1145/3373376.3378452](https://doi.org/10.1145/3373376.3378452).

- [30] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. “PMTTest: A Fast and Flexible Testing Framework for Persistent Memory Programs.” In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 411–425. ISBN: 978-1-4503-6240-5. DOI: [10.1145/3297858.3304015](https://doi.org/10.1145/3297858.3304015).
- [31] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. “ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging.” In: *ACM Trans. Database Syst.* 17.1 (Mar. 1992), pp. 94–162. ISSN: 0362-5915. DOI: [10.1145/128765.128770](https://doi.org/10.1145/128765.128770).
- [32] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnappalli, Pandian Raju, and Vijay Chidambaram. “Crash-Monkey and ACE: Systematically Testing File-System Crash Consistency.” In: *ACM Trans. Storage* 15.2 (Apr. 2019). ISSN: 1553-3077. DOI: [10.1145/3320275](https://doi.org/10.1145/3320275).
- [33] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. “AG-AMOTTO: How Persistent is your Persistent Memory Application?” In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1047–1064. ISBN: 978-1-939133-19-9. URL: <https://www.usenix.org/conference/osdi20/presentation/neal>.
- [34] Yuanjiang Ni, Jishen Zhao, Daniel Bittman, and Ethan Miller. “Reducing NVM Writes with Optimized Shadow Paging.” In: *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. Boston, MA: USENIX Association, July 2018. URL: <https://www.usenix.org/conference/hotstorage18/presentation/ni>.
- [35] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. “Memory persistency.” In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014, pp. 265–276. DOI: [10.1109/ISCA.2014.6853222](https://doi.org/10.1109/ISCA.2014.6853222).
- [36] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. “All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications.” In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 433–448. ISBN: 9781931971164.
- [37] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. “Persistency Semantics of the Intel-X86 Architecture.” In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: [10.1145/3371079](https://doi.org/10.1145/3371079).
- [38] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. “Can Applications Recover from fsync Failures?” In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 753–767. ISBN: 978-1-939133-14-4. URL: <https://www.usenix.org/conference/atc20/presentation/rebello>.
- [39] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. “Programming for Non-Volatile Main Memory Is Hard.” In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. APSys '17. Mumbai, India: Association for Computing Machinery, 2017. ISBN: 978-1-4503-5197-3. DOI: [10.1145/3124680.3124729](https://doi.org/10.1145/3124680.3124729).
- [40] Steve Scargall. “Introduction to Persistent Memory Programming.” In: *Programming Persistent Memory: A Comprehensive Guide for Developers*. Berkeley, CA: Apress, 2020. ISBN: 978-1-4842-4932-1. DOI: [10.1007/978-1-4842-4932-1](https://doi.org/10.1007/978-1-4842-4932-1).
- [41] Margo Seltzer, Virendra Marathe, and Steve Byan. “An NVM Carol: Visions of NVM Past, Present, and Future.” In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 15–23. DOI: [10.1109/ICDE.2018.00011](https://doi.org/10.1109/ICDE.2018.00011).
- [42] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. “X86-TSO: A Rigorous and Usable Programmer’s Model for X86 Multiprocessors.” In: *Commun. ACM* 53.7 (July 2010), pp. 89–97. ISSN: 0001-0782. DOI: [10.1145/1785414.1785443](https://doi.org/10.1145/1785414.1785443).
- [43] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1558609105.
- [44] Denys Vlasenko. *BusyBox*. URL: <https://busybox.net> (visited on 2022-01-10).
- [45] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. “Aerie: Flexible File-System Interfaces to Storage-Class Memory.” In: *Proceedings of the Ninth European Conference on Computer*

Systems. EuroSys '14. Amsterdam, The Netherlands: Association for Computing Machinery, 2014. ISBN: 9781450327046. DOI: [10.1145/2592798.2592810](https://doi.org/10.1145/2592798.2592810).

- [46] Haris Volos, Andres Jaan Tack, and Michael M. Swift. “Mnemosyne: Lightweight Persistent Memory.” In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 91–104. ISBN: 9781450302661. DOI: [10.1145/1950365.1950379](https://doi.org/10.1145/1950365.1950379).
- [47] Hu Wan, Youyou Lu, Yuanchao Xu, and Jiwu Shu. “Empirical study of redo and undo logging in persistent memory.” In: *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 2016, pp. 1–6. DOI: [10.1109/NVMSA.2016.7547178](https://doi.org/10.1109/NVMSA.2016.7547178).
- [48] Xiaojian Wu and A. L. Narasimha Reddy. “SCMFS: A file system for Storage Class Memory.” In: *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–11.
- [49] Jian Xu and Steven Swanson. “NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories.” In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 323–338. ISBN: 978-1-931971-28-7. URL: <https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu>.
- [50] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. “NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System.” In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 478–496. ISBN: 978-1-4503-5085-3. DOI: [10.1145/3132747.3132761](https://doi.org/10.1145/3132747.3132761).
- [51] Jian Yang, Joseph Izraelevitz, and Steven Swanson. “Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks.” In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 221–234. ISBN: 978-1-939133-09-0. URL: <https://www.usenix.org/conference/fast19/presentation/yang>.
- [52] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. “An Empirical Guide to the Behavior and Use of Scalable Persistent Memory.” In: *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 169–182. ISBN: 978-1-939133-12-0. URL: <https://www.usenix.org/conference/fast20/presentation/yang>.
- [53] Junfeng Yang, Can Sar, and Dawson Engler. “EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors.” In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. Seattle, WA: USENIX Association, Nov. 2006. URL: <https://www.usenix.org/conference/osdi-06/explode-lightweight-general-system-finding-serious-storage-system-errors>.
- [54] Takeshi Yoshimura, Tatsuhiro Chiba, and Hiroshi Horii. “EvFS: User-level, Event-Driven File System for Non-Volatile Memory.” In: *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. Renton, WA: USENIX Association, July 2019. URL: <https://www.usenix.org/conference/hotstorage19/presentation/yoshimura>.
- [55] Baoquan Zhang and David H. C. Du. “NVLSM: A Persistent Memory Key-Value Store Using Log-Structured Merge Tree with Accumulative Compaction.” In: *ACM Trans. Storage* 17.3 (Aug. 2021). ISSN: 1553-3077. DOI: [10.1145/3453300](https://doi.org/10.1145/3453300).
- [56] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. “Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks.” In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 207–219. ISBN: 978-1-939133-09-0. URL: <https://www.usenix.org/conference/fast19/presentation/zheng>.



AINiCo: SmartNIC-accelerated Contention-aware Request Scheduling for Transaction Processing

Junru Li Youyou Lu* Qing Wang Jiazhen Lin Zhe Yang Jiwu Shu

*Department of Computer Science and Technology, Tsinghua University
Beijing National Research Center for Information Science and Technology (BNRist)*

Abstract

High-performance transaction processing needs to schedule numerous requests from the network. However, such request scheduling comes with costs of complex information gathering and considerable computation. We observe that emerging SmartNICs pose opportunities for transaction scheduling with low overhead. In this paper, we propose AINiCo, which leverages SmartNICs to intelligently schedule incoming transaction requests to CPU cores, minimizing inter-transaction contention with low latency. AINiCo describes the contention according to system states in a way that SmartNICs can efficiently process, and co-designs hardware and software to enable flexible and adaptive scheduling. We implement AINiCo using FPGA-equipped Innova-2 SmartNICs, and our evaluation shows that AINiCo boosts the throughput by $1.30\times \sim 2.68\times$ and reduces the latency by up to 48.8%.

1 Introduction

Transaction processing is a critical building block for many applications such as e-commerce and the stock exchange. Over the last few years, two trends in transaction processing stand out. First, network bandwidth has improved significantly, enabling a surge of transaction requests from the network. Second, modern servers are equipped with numerous CPU cores, providing abundant computing capacity for transaction processing [1–4]. These two trends together pose a crucial problem: *how to schedule each individual transaction request to the most appropriate CPU core?* Using a sophisticated scheduler for transaction processing can mitigate lots of inter-transaction contention (i.e., two transactions concurrently access the same record, and at least one performs a write), reducing transaction aborts/blocking and thus boosting system throughput.

There are two setbacks in realizing efficient transaction scheduling. First, it is hard to gather necessary information

for scheduling in consideration of cost-efficiency. This is because the transaction system is intricate: each request contains a write/read-set with multiple records, and different records have different degrees of hotness [5–7] and affinities. The hotness and affinity are dynamic: for example, in the live selling platform of Kuaishou, product popularity changes over time due to the behavioral uncertainty of users. Second, transaction scheduling incurs considerable computation overhead. For an incoming request, to calculate which CPU core it executes on causes the least contention, the scheduler has to consume computation cycles that are proportional to the number of CPU cores and the request’s write/read-set size.

We observe that exploiting emerging SmartNICs [8, 9] can enable efficient transaction scheduling for two reasons. First, every transaction request/response flows through NICs, so they are the natural place to gather information about scheduling. Second, transaction scheduling requires considerable computation, and SmartNICs are equipped with specialized programmable hardware (e.g., FPGAs), which is adept in fine-grained parallelism. Thus, SmartNICs can serve numerous concurrent scheduling tasks with low latency.

It is non-trivial to design transaction scheduling using SmartNICs. First, we need to describe inter-transaction contention in a hardware-friendly manner so as to squeeze out the parallelism capabilities of the specialized hardware in SmartNICs. Second, it is impossible for SmartNICs to perform all scheduling tasks since CPU-side transaction software owns important information about transaction execution, e.g., the abort rate, which is indispensable for scheduling. Thus, to enable effective transaction scheduling, SmartNICs should cooperate with upper-level transaction software.

We propose AINiCo, a transaction processing system with on-NIC request scheduling. At the core of AINiCo is a SmartNIC-accelerated contention-aware scheduler; it schedules incoming transaction requests to different CPU cores intelligently, minimizing contention between transactions with low latency.

To describe the inter-transaction contention in a hardware-friendly manner, we first classify system runtime states into

*Youyou Lu is the corresponding author (luyouyou@tsinghua.edu.cn).

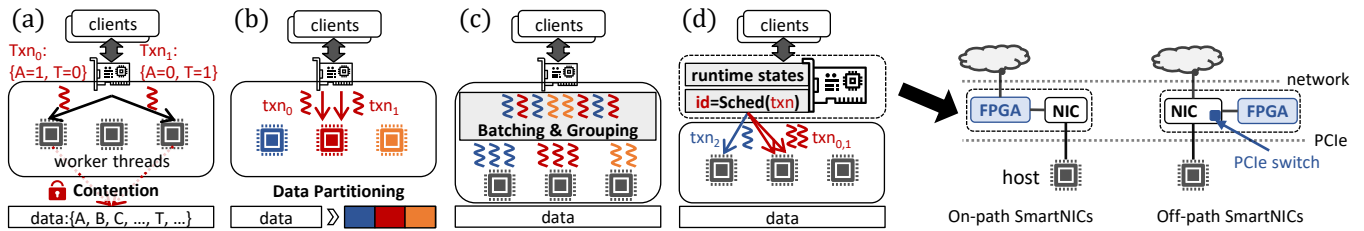


Figure 1: **The architectures of networked transaction systems (a-d).** (a): A system without scheduling. (b): A system using the static data partitioning method. (c): A system using the batching-based scheduling method. (d): AINiCo.

three types: 1) *request state*, i.e., the set of records accessed by a request and the associated access modes (read and write); 2) *worker state*, i.e., the set of requests being executed by the worker thread on each CPU core; 3) *global state*, i.e., workload characteristics such as hotspots. Then, we encode the above three types of states into compact vectors and translate contention calculation into fast vector computation on SmartNICs. For an incoming request, to calculate its contention against different CPU cores, the scheduler compares the request state and each worker’s worker state, with the global state as the weight. We optimize the calculation process via fine-grained parallelism, ensuring low scheduling latency.

To enable adaptive scheduling for time-varying applications, AINiCo adopts a feedback mechanism from the upper-level transaction software to the SmartNICs. Specifically, software in AINiCo periodically samples the global state as the scheduling guideline and updates the on-NIC scheduler by flexible hardware/software interfaces. In this way, AINiCo can handle dynamic workloads in which hotspots change over time. In addition, via the generalized feedback mechanism, AINiCo can support various concurrency control protocols.

We implement AINiCo with Mellanox InnoVA-2 [8], a SmartNIC equipped with an FPGA. The evaluation shows that AINiCo reduces the abort rate by contention-aware transaction scheduling. AINiCo boosts the throughput by $1.30\times \sim 2.68\times$ than the original system with various concurrency control protocols. In addition, compared with existing static data partitioning methods, AINiCo can handle time-varying and skewed workloads; compared with batching-based scheduling methods, AINiCo schedules requests in real-time and renders latency two orders of magnitude lower.

In summary, we make the following contributions:

- A SmartNIC-enabled contention-aware transaction scheduler that offers low overhead by exploiting the acceleration provided by the FPGA.
- AINiCo, a transaction processing system based on the scheduler, supporting various concurrency control protocols and time-varying workloads.
- An AINiCo prototype on real hardware, exhibiting higher throughput, lower latency, and less abort rate.

2 Background and Motivation

2.1 Contention-aware Scheduling

In an in-memory single-server transaction system, as shown in Figure 1 (a), the NIC receives transaction requests and dispatches them to worker threads. Each transaction request often includes multiple write/read operations to the in-memory records. Worker threads execute transactions using various concurrency control protocols to guarantee transaction isolation. However, concurrent transactions are more likely to be aborted or blocked when there is contention (i.e., two transactions concurrently access the same record, and at least one performs a write). Aborts result in costly retries, and blocking may be cascaded, both of which degrade performance [3]. Therefore, the transaction system needs to minimize contention among concurrent transactions [10–22]. Designing a contention-aware scheduler that schedules conflicting transactions to the same worker threads is meaningful. However, there is an accuracy-overhead trade-off in designing it.

Accuracy-overhead trade-off. A high-accuracy scheduler can assist the majority of requests in selecting an appropriate (i.e., with less contention) worker thread to execute on. To achieve such accuracy, the scheduler incurs much computation overhead. Unlike single-key get/put requests in key-value stores, transaction requests are more sophisticated, containing multiple operations (reads, writes, and range queries) to multiple records. The packet steering method [23, 24] based on RSS [25] or Flow Director [26], which embeds a single key in the packet header as the request dispatching information, is no longer effective for transaction requests. Furthermore, the records have different degrees of hotness (i.e., a small set of records are accessed frequently) and affinities (i.e., two records are often accessed together). The hotness and affinity are also constantly changing. As a result, contention-aware scheduling is time-consuming and induces high overhead for high accuracy.

2.2 Existing Scheduling Methods

We revisit two transaction scheduling methods and show their choices in the accuracy-overhead trade-off.

Static data partitioning method. As shown in Figure 1 (b), recent studies [10–12] maintain a data partition scheme, i.e., a mapping from records to worker threads. Clients know the partition scheme and send transactions directly to worker threads without scheduling. Then, each worker thread executes the transactions that access records in a single partition sequentially. Further, Jepsen et al. [15, 16] use a programmable switch to triage transactions belonging to different data partitions before sending them to the database server.

This method is effective for partitionable workloads where transactions tend to access records in a single partition because it has no scheduling overhead under such workloads. However, this method sacrifices the accuracy for two types of workloads: ❶ workloads that do not have a good partition scheme and ❷ workloads whose record affinities and degrees of hotness are constantly changing. In the first type of workloads, many transactions are cross-partition. These transactions require synchronization between worker threads to access remote partitions. In the second type of workloads, the partition scheme needs to be updated to follow workload changes. Recent studies all use offline computation on workload traces to generate the partition scheme; therefore, this method can not react to the changes in workloads in time and introduces load imbalance when hotspots change.

Batching-based scheduling method. As shown in Figure 1 (c), in this method, worker threads collect a batch of transactions and divide the batch into n groups (n is the worker thread count), intending to minimize contention between groups. After that, each worker thread executes a transaction group with almost no contention. Specifically, recent studies on this method [18–21] use a *graph partition* algorithm that treats transactions as nodes, contention between them as edges, and transaction groups as sub-graphs.

This method is adaptive and can react to changes in workloads. Moreover, it avoids load imbalance by guaranteeing even grouping (i.e., high accuracy). However, it introduces high latency for batching (i.e., high overhead). For example, in a state-of-the-art system, Strife [21], the batch size is 10K, which adds about 5ms latency for requests. This is insufferable for *in-memory* transaction systems [22] which usually have microsecond-level latency.

2.3 Scheduling with SmartNICs

We observe that exploiting emerging SmartNICs (as shown in Figure 1 (d)) is promising to break the accuracy-overhead trade-off due to the following two advantages.

First, every transaction request/response flows through the NIC, so the NIC is the natural place to implement a scheduler, which can route packets to any worker thread and keep track of running/queuing transactions on each worker thread. Then, with this piece of information, NICs have an opportunity to make accurate and adaptive scheduling decisions. Recent studies [23, 27–29] leverage the NIC as a scheduler to dispatch key-value requests, addressing the load imbalance or head-of-

line blocking problems. However, they do not take transaction semantics into consideration.

Second, FPGA-equipped SmartNICs can perform lots of computation to generate accurate scheduling decisions with low latency. FPGA-equipped SmartNICs allow users to customize network processing logic. Also, the FPGA modules can reduce scheduling overhead with hardware acceleration instead of amortizing it by batching transactions. Recent studies [30–36] also show that packet manipulation processes with data/pipeline parallelism (i.e., encryption and serialization) are suitable to be offloaded to the FPGA modules on SmartNICs. Such offloading can accelerate networking and relieve the host-side CPU/memory burdens.

FPGA-equipped SmartNICs can be divided into two categories, on-path and off-path, as shown on the right of Figure 1. They vary in the connection architecture between the FPGA and other NIC components. In an *on-path* SmartNIC, the FPGA is located between network ports and the NIC ASIC. In this type of SmartNICs, the FPGA modules need to process all link-layer network traffic, complicating the FPGA logic. Contrarily, in an *off-path* SmartNIC, the NIC ASIC is the same as that of a standard NIC, and packets are routed to either the host or the FPGA by an on-board PCIe switch. We use an off-path SmartNIC, *Mellanox Innova-2* [8], so that we can focus on the scheduling logic while leaving sophisticated network functions (e.g., reliable delivery and ordering) to the full-fledged NIC ASIC.

Mellanox Innova-2 [8] has a ConnectX-5 NIC ASIC and a Xilinx XCKU15P FPGA. It has two 25Gbps ports and uses PCIe 3.0 $\times 8$ to connect the NIC ASIC, the FPGA, and the CPU. The CPU communicates with the FPGA via MMIO (memory-mapped IO) or the FPGA’s DMA engine. The ConnectX-5 NIC ASIC supports RDMA (remote direct memory access), through which remote clients can read/write the server’s memory while bypassing the server’s CPUs.

Challenges of transaction scheduling with SmartNICs. It is non-trivial to design a transaction scheduler using SmartNICs due to the following challenges: 1) The FPGA is adept in fixed and simple execution flows. We need to describe complicated inter-transaction contention in a hardware-friendly manner to squeeze out the FPGA’s parallelism capabilities. 2) Worker threads have important information about transaction execution, e.g., abort rate, which is indispensable for scheduling. Thus, to enable effective transaction scheduling, worker threads should gather this information as feedback to adjust the scheduler. 3) Transaction systems use various concurrency control protocols and face various workloads; however, the FPGA redesign is costly. It is difficult for an application-specific FPGA-based scheduler to support various transaction systems [37, 38]. To be generalized for all transaction systems, the scheduler design (e.g., request format, scheduling algorithm, and feedback interfaces) should not encode any application-specific characteristics.

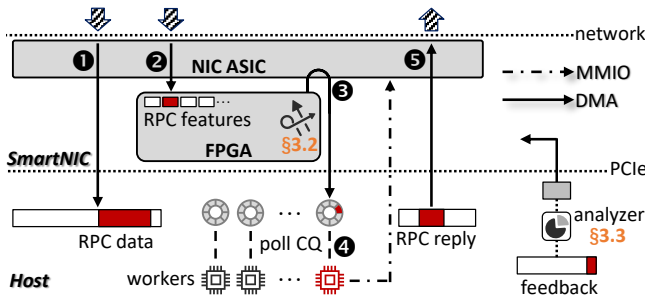


Figure 2: A scheduling-enabled RPC with SmartNICs.

3 AINiCo

3.1 Overview

To reduce the inter-transaction contention and break the accuracy-overhead trade-off, we propose AINiCo, a SmartNIC-accelerated contention-aware request scheduler. AINiCo uses hardware acceleration of the FPGA for low overhead and provides generalized software feedback interfaces for high accuracy.

Transaction systems typically provide a *stored procedure* interface, where each type of transaction is compiled into a procedure, and clients issue transactions via remote procedure calls (RPCs). To achieve request scheduling, SmartNICs need to parse the procedure parameters and then detect inter-transaction contention. However, the parameters are application-specific, including keys that the transaction will access, the access modes (read/write), the values that the transaction will write, and so on. The format of the parameters also varies across applications.

Therefore, in order to support diverse applications without encoding application-specific properties, we include a fixed-form header in each request. Clients need to convert the parameters into the fixed-form header. We call this the *request feature vector*. Scheduling also needs the information on running/queuing transactions in worker threads and workload characteristics. AINiCo encodes them in a hardware-friendly manner and leverages the data and pipeline parallelisms of the FPGA to make scheduling decisions (§3.2).

We design a **scheduling-enabled RPC** with SmartNICs as the communication mechanism between clients and the server, allowing the FPGA to receive requests and schedule them. Figure 2 describes its workflow. The server maintains a data buffer on the host memory and a feature vector buffer on the FPGA. To issue a transaction, the client sends the data (1) and the feature vector (2) to their respective buffers via two one-sided RDMA writes posted together. AINiCo’s scheduler (on the FPGA) polls the feature vector buffers to determine the arrival of new requests. The scheduler then selects the most appropriate worker thread based on the request’s feature vector, the runtime states of worker threads, and the workload characteristics. After making the scheduling decision, the

scheduler notifies the selected worker thread via writing its receive completion queue (CQ) (3). The CQ entry includes only the address of request data but not the feature vector. Since RDMA enforces ordered writes, after reading a new CQ entry (4), the worker thread can get the completed data from the RPC data buffers. The worker thread then executes the transaction and replies to the client via a normal RDMA write (5).

To make the scheduler adaptive to workload changes, AINiCo employs a software feedback mechanism (§3.3). It allows the software to use information about transaction execution (e.g., records that cause transactions to be aborted) to guide the scheduler. The feedback interface is generalized for various concurrency control protocols and different workloads. The FPGA fetches the feedback periodically via DMA reads outside the critical path.

3.2 Accelerated Scheduling on The Hardware

To describe the inter-transaction contention and schedule transactions in a hardware-friendly manner, we identify three types of states for scheduling: *request state* (§3.2.1), *worker state* (§3.2.2), and *global state* (§3.2.3). We then encode the states into compact *vectors* and translate the scheduling algorithm into the fast *vector computation*, which is well-suited for the FPGA (§3.2.4).

3.2.1 Request state

AINiCo uses the request state to describe the resources required by a transaction. Clients embed the complex parameters of a transaction request into a request feature vector. Specifically, each feature vector (f_{req}) has L elements¹. We use a mapping function to divide all records into L *groups*. Each element in the vector represents whether the transaction will access the corresponding group of records. The mapping function is the same in all clients. To generate the feature vector, clients enumerate the keys² in the request parameters and set the corresponding elements. Each element can be encoded into either 1-bit or 2-bit element. A 1-bit element describes two access modes: no operation and access (read or write). A 2-bit element describes three access modes: no operation, read, and write, having better expressibility but consuming more space. The scheduler can use the feature vectors of requests to estimate whether contention exists between them. For example, if two requests set the same element in their feature vectors, these two requests might have contention.

The mapping function for generating feature vectors. The mapping function is used to select a group for a given key. Clients use a hash function to calculate a hash value (v) for each key and use $v \% L$ as the group number. Due to hash collisions, clients might map two different keys to the same group.

¹All vectors have a length of L in this paper.

²The primary keys of records.

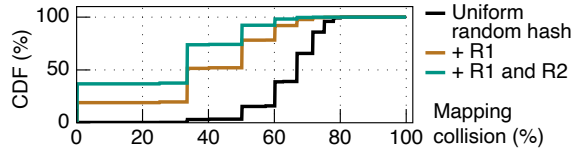


Figure 3: Mapping collisions of two New-order txns.

To reduce the negative impact of hash collisions, AINiCo imposes two requirements (**R**) on the mapping function. First, transaction systems manage records with different semantics in different tables (e.g., the CUSTOMER table and the ORDER_LINE table in TPC-C) and access records through $\langle table_id, key \rangle$. Therefore, clients should map the keys of different tables into different groups (**R1**). For example, if an application has two tables, the mapping function maps the keys of one table to $\frac{1}{2}$ groups and maps the keys in the other table to the other $\frac{1}{2}$ groups. Second, because different tables can have different sizes (i.e., numbers of records), the number of groups of a table should be proportional to its size (**R2**). We test the CDF of the mapping collisions between New-order transactions in TPC-C, as shown in Figure 3. The mapping function with these two requirements reduces the mapping collisions compared with the simple uniform random hash. Note that when the table count is large, all tables should instead share the feature space since there are not enough elements in the feature vector.

Discussion. There are two corner cases worth discussing when describing the request state.

How to handle data growth. In our current design, the feature vector length L and the key-to-feature mapping function are static at runtime. When the amount of data grows, AINiCo does not need to increase L correspondingly in most cases. This is because the determinant of mapping collisions is the amount of hot data instead of the data in the entire system; only a small amount of data is accessed frequently. However, if changing L or the mapping function is a must (e.g., when the hotspot size changes significantly), AINiCo can reload the on-NIC scheduler with the new configuration (which takes 4.18ms on InnoVA-2 in our evaluation). In that case, clients also need to update the new configuration.

How to describe pre-unknown keys. Clients specify most of the keys of the records that a transaction will access in the request’s parameters. However, some keys are pre-unknown and can only be determined through transaction execution. To describe the required resources for these transactions more accurately, AINiCo employs the following two speculative optimizations. First, for transactions containing range queries, clients randomly generate the keys in the ranges and set the corresponding elements of the request feature vector. Second, for transactions that access records via non-primary keys, the client maintains a secondary-index cache, which maps each non-primary key to a set of primary keys, to determine the records that those transactions will access.

	no op	read	write	read & write
request feature vector f_{req}^{\rightarrow}	00	10	01	01
worker feature vector f_w^{\rightarrow}	00	01	11	11

Table 1: Encoding of read and write modes.

3.2.2 Worker state

AINiCo uses the worker state to describe the resources that are being or will be accessed by worker threads. Each worker thread has different running/queuing transactions. Therefore, a state of the i -th worker thread is the union of the feature vectors of its running/queuing transactions. We call it the **worker feature vector** ($f_{w_i}^{\rightarrow}$ for the i -th worker). This vector has the same format as the request feature vector. Further, to allow worker threads to steer requests actively, AINiCo introduces the other worker state, called the **worker steering vector**. Each worker thread maintains a steering vector with 1-bit elements ($s_{w_i}^{\rightarrow}$ for the i -th worker). Worker threads can set their steering vectors via the software feedback interface, which is described in detail in §3.3.2.

How to describe the contention. AINiCo uses the bitwise AND (&) operation to describe the contention between a new request and the running/queuing requests in a worker. When the element in the feature vector is 1-bit, AINiCo encodes request features and worker features in the same way, where 1 represents access (read or write) and 0 represents no operation. The AND operation between request features and worker features estimates all concurrent accesses to the same group. When the element is 2-bit, AINiCo encodes the access modes in request features and worker features in different ways, as shown in Table 1. The result of AND operation between the request features and the worker features describes read/write and write/write accesses to the same group as contention.

How to maintain worker feature vectors. The scheduler maintains a feature vector queue (FVQ) for each worker. The FVQ stores the feature vectors of the worker’s running/queuing requests (f_{req}^{\rightarrow}), in the same order as the requests in the worker’s CQ. The new worker feature vector is recalculated on adding/deleting an f_{req}^{\rightarrow} to/from the worker’s FVQ. Specifically, for each worker, the scheduler keeps two counter vectors. A counter vector counts the number of read requests on different groups, and the other vector counts the write requests. When a request is added/deleted, the scheduler updates the counter vectors and then converts them to the aforementioned worker feature vector with the FPGA’s parallelism.

When a request completes, the scheduler needs to remove its feature vector from the FVQ. One naive method is to let the worker thread notify the scheduler after executing the request. However, it requires an additional MMIO operation, which may slow down the system. AINiCo designs a lazy updating mechanism. Specifically, the scheduler will check the completed requests only when the scheduler pushes a new request to a worker’s CQ. This can be done efficiently through the FPGA’s DMA without invoking the CPU.

State of	Name	Format	Maintainer
request	feature: f_{req}^{\rightarrow}	vector: 2-bit \times L	client
worker	feature: f_w^{\rightarrow}	vector: 2-bit \times L	scheduler
	steering: s_w^{\rightarrow}	vector: 1-bit \times L	
global	weight: \vec{W}	vector: 8-bit \times L	worker thread
	worker-set table	type \rightarrow worker set	

Table 2: The states used for making scheduling decisions.

3.2.3 Global state

AlNiCo uses the global state to describe workload characteristics. Real-world applications have skewed access patterns, where some records are accessed frequently (i.e., hotspots). These hot records are the main source of contention. Therefore, in AlNiCo, one of the global states is the **weight vector**. Each element in the vector has 8 bits and is the weight of a group. The element actually represents the total sum hotness of the keys in a group. To make our scheduler adaptive to hotspot changes, the weight vector is dynamically updated by the software, and the feedback interface is detailed in §3.3.1.

The other global state is the **worker-set table**, which stores a set of worker threads for each request type. The scheduler selects a worker only from each request type’s worker set. This is used to avoid the head-of-line blocking for long-running requests, which is described in detail in §3.3.3.

3.2.4 Making scheduling decisions

As summarized in Table 2, states are formatted in vectors with length L, except for the global worker-set table. Therefore, the scheduler can use fast vector computation to make scheduling decisions. For each new request, the scheduler performs the following three steps.

Step#1. The scheduler searches the worker-set table to get the set of workers for this type of transaction.

Step#2. For each worker, the scheduler calculates a **contention rank** ($rank_{w_i}$ for the i-th worker) between the new request and the running/queuing requests in the worker. The contention rank is calculated using the following formula:

$$rank_{w_i} = (\text{sign}(f_{req}^{\rightarrow} \text{ AND } f_{w_i}^{\rightarrow}) \text{ AND } s_{w_i}^{\rightarrow}) \text{ DOT } \vec{W}$$

Here we assume elements in steering vectors ($s_{w_i}^{\rightarrow}$) are 1. When each element in feature vector is 2-bit, AlNiCo needs the **sign** function to transform the result of ($f_{req}^{\rightarrow} \text{ AND } f_{w_i}^{\rightarrow}$) to a vector with 1-bit elements for later calculations. If the input is greater than 0, the result of the **sign** function is 1, otherwise it is 0. This result res_i^{\rightarrow} ³ represents the same groups accessed by the new request (f_{req}^{\rightarrow}) and the i-th worker’s running/queuing requests ($f_{w_i}^{\rightarrow}$). The contention rank of the i-th worker is the weighted sum (\vec{W}) of res_i^{\rightarrow} .

³We denote the results of $\text{sign}(f_{req}^{\rightarrow} \text{ AND } f_{w_i}^{\rightarrow}) \text{ AND } s_{w_i}^{\rightarrow}$ as res_i^{\rightarrow} .

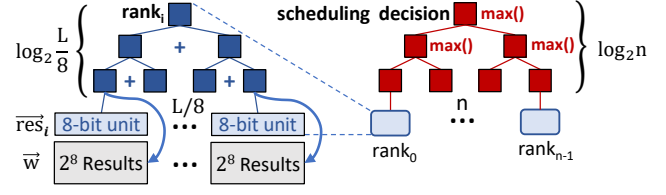


Figure 4: Hardware acceleration for scheduling.

Step#3. The scheduler selects the worker with the *highest* contention rank among n workers⁴ to route the request.

Hardware acceleration. We leverage the FPGA to accelerate the scheduling calculation, including the AND (&) and DOT PRODUCT (·) to compute the contention rank of each worker, and the MAX to select a worker with the highest rank.

(1) We leverage the FPGA data parallelism to calculate all bits simultaneously during the AND operation.

(2) Because each element in res_i^{\rightarrow} is 1-bit, the DOT PRODUCT operation between res_i^{\rightarrow} and \vec{W} is actually a summation operation, which adds a weight value into the sum only if the corresponding bit in res_i^{\rightarrow} is 1. We optimize the summation via a binary tree reduction algorithm. As illustrated in Figure 4, the reduction algorithm executes the + operation in parallel using many computation units. Each unit adds two values in the L inputs, and then the $\frac{1}{2}$ results become new inputs. It repeats this process until there is only one result left. This algorithm only needs to execute $\log_2 L$ times but uses $\frac{1}{2}$ computation units. To save the computation units, we trade memory consumption. Because the weight vector is periodically updated, it is constant for a period. The FPGA stores the results of the DOT PRODUCT between different res_i^{\rightarrow} and the constant \vec{W} in advance. However, res_i^{\rightarrow} has 2^L cases, and it is impractical to store all results. Therefore, we divide res_i^{\rightarrow} into $\frac{1}{8}$ segments. Each segment includes 8 bits and has $2^8 = 256$ results. There are $\frac{1}{8} \times 2^8$ results in total, small enough to be stored in the FPGA’s memory. The FPGA still uses the reduction algorithm to compute the sum of these segments’ results. Storing segment result trades the memory for reducing the number of computation units from $\frac{1}{2}$ to $\frac{1}{16}$.

(3) The MAX operation is to find the highest value among n values. As shown in Figure 4, the FPGA uses the MAX reduction algorithm, which requires only $\log_2 n$ steps.

These hardware optimizations are leveraged to reduce the overhead of making accurate scheduling decisions.

3.3 Adaptive Feedback from The Software

To make the scheduler adaptive to the changes in workload characteristics, AlNiCo allows the software to set the following three states: the weight vector \vec{W} (§3.3.1), the workers’ steering vectors s_w^{\rightarrow} (§3.3.2), and the worker-set table (§3.3.3). AlNiCo provides generalized feedback interfaces to worker threads and uses a lightweight *analyzer thread* to update the

⁴We denote n as the worker count in the following paper.

```

1 void hotness_feedback(i: worker id, key){
2   hotness[i][hash(key)]++;
3 }
4
5 void affinity_feedback(group_A, group_B){
6   set_affinity(group_A, group_B)
7 }
8
9 void worker_set_feedback(txn_type, worker_set);
10
11 void update_weight(E: epoch id){ // Sec.3.3.1
12   TP(E) =  $\sum_{i=0}^{n-1} tp_i$ 
13    $\vec{h}_{all} = \sum_{i=0}^{n-1} \vec{h}_i \times tp_i / TP(E)$ 
14   if ( E == 0 || TP(E) < (1-c) × TP(E-1) )
15     update  $\vec{w}$  based on  $\vec{h}_{all}$  // normalization
16   next_epoch(): clear  $tp_i$  and  $\vec{h}_i$ 
17 }
18
19 void update_steering( $\vec{w}$ ){ // Sec.3.3.2
20   g_groups = [] // the indexes of guideline groups
21   for weight in sorted  $\vec{w}$  { // from the highest
22     new_group = weight.index;
23     if no_affinity(new_group, g_groups)
24       g_groups.append(new_group);
25     if (g_groups.size() == n) break;
26   }
27   update steering vectors based on g_groups
28 }

```

Listing 1: The feedback interfaces and algorithms.

scheduler’s states based on the feedback. Listing 1 shows the feedback interfaces and the algorithms for translating the feedback into the states needed by the scheduler.

3.3.1 Hotness feedback

We define the keys that *cause contention frequently* as the hotspots in the transaction system, instead of the keys that are accessed frequently. The hotspots keep changing over time. AINiCo requires the software to identify the hotspots in real-time and update the weight vector correspondingly.

Hotness feedback interface. We provide a hotness feedback interface as shown in Lines 1-3. Each worker increases the hotness of a key in its exclusive hotness vector (\vec{h}_i) without any coordination with other workers. Different concurrency control protocols invoke the interface in different situations, and we divide them into two cases based on the way they address the contention. First, with OCC and the 2PL that avoids deadlock by wait-free algorithms, transactions might get aborted and re-executed. The worker increases the hotness of the *record that causes the abort*. Second, with the 2PL that holds locks sequentially to avoid deadlock, transactions might be blocked while trying to acquire a lock. Every time a transaction *retries acquiring the lock of a record*, the worker increases the corresponding hotness.

How to update the weight vector. AINiCo employs an epoch-based updating approach. The analyzer thread collects these hotness vectors at the end of each epoch. Further, each worker measures its throughput (tp_i). Then, as shown in Line 13, the analyzer calculates the weighted average (based on tp_i) of workers’ hotness vectors as the global hotness vector (\vec{h}_{all}) and uses it to update the weight vector (Line 15).

When to update the weight vector. AINiCo updates the weight vector only when the real hotspots change. When the system is cold-started (e.g., in epoch 0), requests are randomly scheduled, and therefore \vec{h}_{all} can reflect the real hotspots. However, when the system is stable, due to AINiCo’s effective scheduling mechanism, those hot keys detected right after the cold start no longer cause contention, and their degrees of hotness is low in the new \vec{h}_{all} . In this case, \vec{w} should remain unchanged to keep the scheduler effective.

The software uses throughput decreases or abort rate increases as the signal of the hotspot changes. Line 14 shows how to detect changes in hotspots. In this algorithm, we use the throughput (TP) decreases as the signal. If the throughput decreases significantly, i.e., TP drops by more than a factor of c , AINiCo will store \vec{h}_{all} in a state buffer that can be read by the NIC via DMA.

When the workload is read-intensive, workers increase the hotness every time a record is read. The analyzer simply ignores the workload change signal and always updates the weight vector at the end of epochs. In this way, AINiCo helps worker threads to better leverage cache locality.

3.3.2 Affinity feedback

A contention-aware scheduler not only schedules conflicting transactions to the same workers but also schedules transactions without contention to different workers to make all workers busy. To this end, AINiCo introduces the steering vectors (\vec{s}_w) to guide the scheduler. As shown in the formula in §3.2.4, only if the element in a worker’s steering vector ($s_{w_i}^{\vec{s}}$) is 1, the corresponding element in its feature vector ($f_{w_i}^{\vec{s}}$) is valid. The key idea is to select n (i.e., worker count) **guideline groups** that have the *highest weights* and *do not have affinities with each other*. AINiCo assigns different guideline groups to different workers to achieve the aforementioned goal. Each worker’s steering vector consists of 1-s for its assigned guideline group and all non-guideline groups by default. For other guideline groups, the corresponding element in the steering vector is 0. In this way, the worker will steer transactions accessing its guideline group. Therefore, in a running system, the feature vector of a worker represents the groups that have affinities with the worker’s guideline group.

Affinity feedback interface. To find the n guideline groups, AINiCo needs the affinity characteristics of workloads. We provide the affinity feedback interface (Lines 5-7) that allows users to offer the affinity hint for different workloads.

How to update the steering vectors. As shown in Lines 19-28, the analyzer thread first sorts all groups according to their weights and enumerates them from the group with the highest weight. Then, the analyzer thread checks whether the group has an affinity with the already found guideline groups. If there is no affinity between them, the group will be labeled as a new guideline group.

3.3.3 Reserving workers for long-running transactions

In real-world workloads, some transactions take a long time to finish (e.g., analytical transactions) and block the transactions assigned to the same worker (i.e., head-of-line blocking). For example, a Delivery transaction accesses about 10 times more keys than a New-order transaction. We reserve some workers to handle these long-running transactions, which is similar to the size-aware request scheduling mechanism [39, 40]. We provide an interface to set the worker set for different types of transactions. Our design only sets the worker-set table on the system startup for workloads that have long-running transactions.

4 Implementation

We implement AINiCo on an Innova-2 SmartNIC [8] and use STO [41], a state-of-the-art transaction processing framework, as the backend transaction processing module.

The scheduler on the FPGA. We implement two fundamental drivers for SmartNICs: a PeerDirect driver [42, 43] to build peer-to-peer communication between the NIC ASIC and the FPGA, and a DMA driver [44] to connect the FPGA and the host. Innova-2 holds a Xilinx KU15P FPGA. In our implementation, the feature vector length L is 512, and we encode read/write modes to the same bit to save feature space. The clock frequency is 250MHz. With these configurations, AINiCo consumes 54 cycles for each request to make the scheduling decision. AINiCo updates global states every 1024 requests (every 22ms) in the background, consuming 1184 cycles. Therefore, we set the epoch size for gathering the feedback in software to 20ms. The implementation consumes 159K (30.48%) LUTs, 157K (15.10%) FFs, and 678.5 (68.95%) BRAMs. We evaluate the computation cycles and resource usage with different feature vector lengths in §5.6.

The transaction processing module on the host. We choose STO as our backend transaction processing module because it implements various CC protocols in the same framework and is high-performance. To use STO, we locate the contention handling function in the framework and then add feedback interface codes to these functions. The parameters of this feedback interface are the records that triggered the contention handling function. On the client side, we do not modify the original request format of the stored procedure, and we only add the field for the feature vector in the request header. We use the following four concurrency control (CC) protocols in the STO framework:

- OCC: it is based on Silo [45], which avoids allocating global timestamps to improve multi-core scalability.
- TicToc [46]: it is an OCC variant that assigns commit timestamps dynamically according to read/write set.
- MVCC: it is based on Cicada [47], which optimizes the management of the timestamps and multi-version values.
- 2PL [48, 49]: it uses NO_WAIT to avoid deadlock; when a transaction fails to acquire a lock, it immediately aborts.

All four concurrency control protocols run at the serializable isolation level. The table indexes use Masstree [50], which supports range queries.

5 Evaluation

We evaluate AINiCo under various workloads and seek to answer the following five questions:

1. How does AINiCo perform compared with existing transaction scheduling methods (§5.2)?
2. How does AINiCo react to dynamic workloads (§5.3)?
3. Why are SmartNICs necessary for the contention-aware request scheduling (§5.4)?
4. How compatible is AINiCo with various concurrency control protocols (§5.5)?
5. What are the overhead and the limitation incurred by the on-NIC scheduler in AINiCo (§5.6)?

5.1 Experimental Setup

Experimental environment. We run all experiments on three machines, one as the server and two as clients.

Hardware settings. Each machine is equipped with two 12-core Xeon E5-2650 v4 2.20GHz CPU sockets, PCIe 3.0 interfaces, and memory of 128GB. The database server is equipped with a dual-port 25Gbps Innova-2 SmartNIC. Each client is equipped with one 100Gbps Mellanox ConnectX-5 NIC. They are connected by a 100Gbps Mellanox switch.

Software settings. Each client thread issues up to 4 transaction requests simultaneously (i.e., the queue depth is 4). We adjust the number of clients and the number of asynchronous requests per client to change the test pressure. The server uses 20 cores for worker threads, 2 cores for long-running transactions in TPC-C workloads, and one core for the analyzer thread and performance measurement. For a fair comparison, the competitors use the RDMA-based RPC with optimizations from previous RDMA systems [51–58], and only use the normal ASIC part of the Innova-2 NIC. We evaluate the performance of this RPC against our scheduling-enable RPC in §5.6.

Workloads. We use the following benchmarks:

TPC-C [6] simulates the activity of a wholesale supplier with five types of transactions. We use the full-mix TPC-C. AINiCo reserves two cores for the long-running delivery transactions. **YCSB-T** is a transactional extension of YCSB, which is a popular KV store benchmark [59]. It has 20 tables as many as the worker threads, and the key space is 100M. Each record contains an 8-byte key and a 384-byte value [39, 60]. Each transaction accesses 16 records in a single table, and the keys are generated according to the Zipf distribution ($\theta=0.99$). **YCSB-HOT** is a dynamic workload based on YCSB-T. Different from YCSB-T, it has 100 tables, and 20 tables are hot at one time. The hot tables are changed every two seconds to

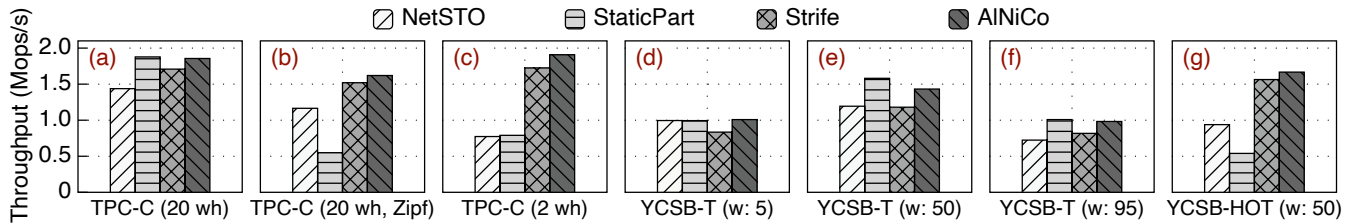


Figure 5: **Throughput.** The *wh* in TPC-C is the number of warehouses, (a): the low-contention workload, (b): the skewed workload under Zipf $\theta=0.99$, (c): the high-contention workload. The *w* in YCSB is the percentage of write operations.

simulate the changes in time-varying applications. Keys are generated according to the Zipf distribution ($\theta=1.2$).

Competitors. We compare AINiCo with five systems: a baseline system without scheduling, two systems using existing scheduling methods, and two CPU-based versions of AINiCo. NetSTO is a baseline system in which clients randomly select a worker to send transaction requests without scheduling. StaticPart is a system using the static data partitioning method. For TPC-C workloads, StaticPart partitions data based on the warehouse ID [12]. For YCSB-T and YCSB-HOT workloads, StaticPart partitions data based on the ID of tables. Strife [21] is a system using the batching-based method. The batch size is 10K, and the batch waiting time is 5ms, the same configuration as in Strife’s paper. To support batching, the queue depth in clients is 1K.

AINiCo-CPU-2 is a CPU-based version of AINiCo, which reserves two dedicated threads to execute the scheduling logic. Each scheduler thread connects to half of the clients and communicates with the workers through separated message queues. The worker/global states are shared by the scheduler threads and updated with atomic operations.

AINiCo-CPU-N is the other CPU-based version of AINiCo. It co-locates the worker logic and scheduler logic in each thread, where they multiplex the CPU resource. All threads share the worker/global states. The client connections are evenly distributed among threads. Each thread can make scheduling decisions and delegate requests to others as a scheduler.

5.2 Overall Performance

We first evaluate the peak throughput of NetSTO, StaticPart, Strife, and AINiCo with various workloads in Figure 5. Then we evaluate the latency of different types of transactions under TPC-C by varying request pressure in Figure 6.

Throughput under TPC-C. We evaluate TPC-C under different levels of contention by varying the number of warehouses. We set the warehouse count to 20 and 2 to simulate low-contention and high-contention scenarios. We also introduce a skewed TPC-C, which has 20 warehouses. In this skewed TPC-C, clients select the warehouse according to the Zipf distribution with $\theta = 0.99$. In StaticPart, when the warehouse count is equal to the worker count (20), each worker manages an exclusive warehouse, and when the warehouse

count is 2, every 10 workers manage a warehouse. For a fair comparison, StaticPart, Strife, and AINiCo all reserve 2 worker threads for long-running transactions. Figure 5 (a)-(c) show the throughput of TPC-C under these configurations, and we have the following two observations.

First, in the low-contention workloads (Figure 5 (a)), StaticPart has the highest throughput, outperforming NetSTO, Strife, and AINiCo by 1.30 \times , 1.11 \times , and 1.06 \times , respectively. This is because only 10% of the transactions are cross-warehouse transactions, and the warehouse count is the same as the worker count, which is a good case for the static data partitioning method. The transaction grouping algorithm in Strife and the feedback mechanism in AINiCo cost the CPU resources that are originally used for transaction execution. In the skewed workloads (Figure 5 (b)), the throughput of StaticPart is only 47% of NetSTO. This is because clients access hot warehouses while the data partition in StaticPart is static, which results in load imbalance.

Second, in the high-contention workloads (Figure 5 (c)), 1) AINiCo improves throughput by 2.46 \times compared with NetSTO. This is because the contention-aware scheduling reduces the contention between the running transactions and saves the CPU resources to execute more transactions. 2) The throughput of AINiCo is 2.41 \times higher than StaticPart. This is because the workload is not partitionable and there is still a lot of contention between concurrent transactions.

Throughput under YCSB-T. Figure 5 (d)-(f) show the throughput under YCSB-T with different read/write ratios. We have the following two new observations.

First, all scheduling methods do not have benefits under the read-intensive workload (Figure 5 (d)). This is because 1) there is less contention for read-intensive transactions; 2) the throughput of this workload is bound by the bandwidth of the NIC (i.e., 50Gbps); 1Mops/s throughput in this workload requires about 45.6Gbps outbound bandwidth to transmit data.

Second, the highest throughput of write-intensive workloads (Figure 5 (f)) is bound by the NIC in AINiCo. However, NetSTO can not use the full bandwidth because this workload causes more contention (especially the write-write contention) than the read-intensive one.

Latency under TPC-C. We evaluate the latency distribution for New-order transactions and Delivery transactions (long-running) with varying throughput. Figure 6 shows the median

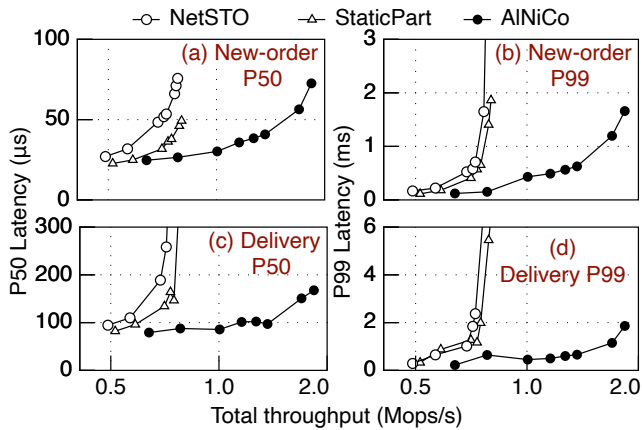


Figure 6: Latency of New-order and Delivery transactions under TPC-C (2 warehouses).

(P50) and 99th percentile (P99) latency, and we have the following two observations.

First, as shown in Figure 6, when the throughput is low, the median latency of New-order in AINiCo ($24.1\mu s$) is higher than that in NetSTO ($23.6\mu s$). This is because the requests in AINiCo have extra latency for scheduling logic and two extra PCIe communication latency for the off-path SmartNICs. As the throughput increases, the New-order median latency in AINiCo is lower than NetSTO because the reserving worker threads for long-running transactions prevent them from blocking other normal transactions.

Second, in NetSTO, before the throughput reaches the peak, the latency increases faster than in AINiCo. This is because the contention blocks transactions or causes them to retry, increasing the latency of the running transactions and blocking subsequent requests. Under a similar throughput of AINiCo (0.78Mops/s) and NetSTO (0.71Mops/s), AINiCo reduces the median latency of New-order from $51.8\mu s$ to $26.5\mu s$.

In addition, median and P99 latency in Strife exceed $7ms$, larger than the batch waiting time ($5ms$). In Strife, the median latency of New-order is between $7.00ms$ and $19.92ms$, and the P99 latency is between $12.54ms$ and $25.18ms$, which are not plotted in the figure to avoid obscuring other results. This is because the end-to-end latency includes the network stack latency, the batch time, the transaction grouping time, and the transaction execution time. AINiCo does not sacrifice the request latency: at the peak throughput, the P99 tail latency of the New-order transaction is no more than $1.17ms$, and the median latency is only $72.7\mu s$.

5.3 Dynamic Workloads

We evaluate AINiCo’s ability to adapt to dynamic workloads by changing the hot tables in YCSB-HOT. We run YCSB-HOT with a 50/50 read/write ratio.

Figure 7 shows the throughput over time, from which we have the following three observations.

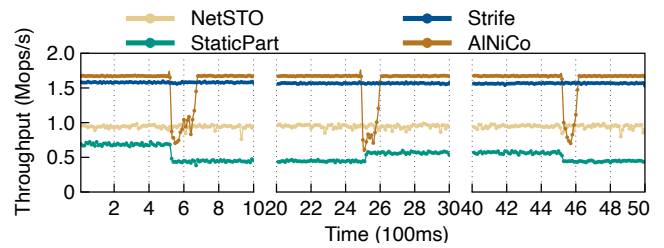


Figure 7: YCSB-HOT throughput over time. Hot tables change every 2s; measuring the throughput every 10ms.

First, AINiCo takes about $150ms$ to adapt to the changes of hot tables. At the beginning of changes, the throughput drops to the lowest point, even worse than NetSTO. This is because the weight vector fails to reflect the conflicting hotspots in the new hot tables. The steering vectors based on the historical hotspots can not guide the scheduling. Before adjusting the scheduler to fit the new workloads, AINiCo suffers performance jitters. This is because the feedback in AINiCo reflects the workload characteristics over time, and it takes a while to make the characteristics of old hot tables fade away. Second, Strife reacts more quickly to dynamic workloads without experiencing throughput degradation. This is because the results of transaction grouping in Strife are based on the information in a batch and do not rely on the historical information of the workloads. However, the median/P99 latency (at the peak throughput) of AINiCo and Strife are $72.6\mu s/1.66ms$ and $12.21ms/15.23ms$, respectively. This is because Strife introduces extra latency due to batching. Third, StaticPart performs worse than NetSTO and varies with the workload changes. This is because the hot tables cause the load imbalance problem in this static data partition method.

In summary, according to §5.2 and §5.3, the static data partition methods are the best for the partitionable workloads, but they can not handle skewed or dynamic workloads. The batching-based scheduling methods can handle various workloads, but they introduce orders of magnitude higher latency for requests. Thanks to SmartNICs, AINiCo can handle skewed or dynamic workloads with low latency.

5.4 Comparison with CPU-based AINiCo

We demonstrate the necessity of SmartNIC-accelerated design for contention-aware scheduling by comparing it with two versions of CPU-based AINiCo.

Figure 8 shows the throughput of CPU-based AINiCo with varying worker thread count. The workload is TPC-C with 2 warehouses. We have the following three observations.

First, AINiCo-CPU-2 brings an improvement in throughput when the worker thread count is small. However, with more worker threads, the performance decreases. This is because two scheduler threads have limited computing resources, and the scheduling complexity increases linearly with the number of worker threads. As a result, the CPU can not accelerate the scheduling computation.

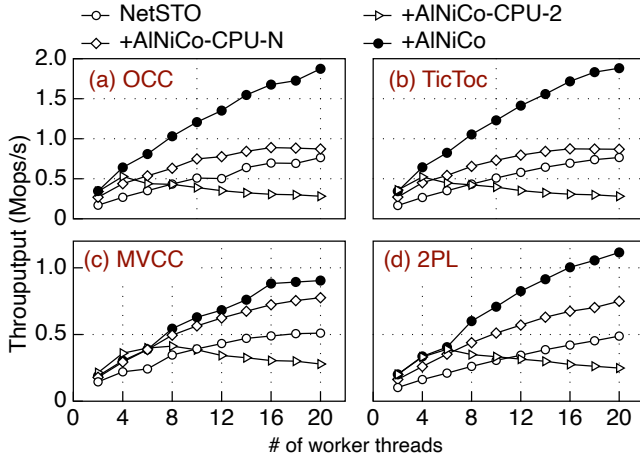


Figure 8: TPC-C (2 warehouses) throughput with four different concurrency control protocols.

Second, AINiCo-CPU-N is not as good as AINiCo-CPU-2 when the worker thread count is small because the scheduling consumes the workers' resources. It improves the throughput compared with NetSTO because its scheduling overhead is less than the overhead for transaction aborts/blocking. Moreover, its throughput is scalable with the thread count because the resources for the scheduler increase linearly with the thread count.

Third, the improvement of the AINiCo-CPU-N version is small. The throughput of the scheduler itself is not the bottleneck in AINiCo-CPU-N, but it can not enjoy the acceleration of the FPGA. The scheduling logic costs lots of CPU resources. AINiCo speeds up individual request scheduling by fine-grained parallel computation, and the computation of multiple requests is pipelined. These provide higher performance with less CPU resource cost.

5.5 Generality of AINiCo

We demonstrate the generality of AINiCo by changing the concurrency control protocols in STO. Figure 8 shows the throughput with varying worker thread count and the four CC protocols described in §4. The workload is TPC-C with 2 warehouses. Note that, since the 2PL implementation in STO has performance abnormalities under full-mix TPC-C, we only evaluate the New-order transactions for 2PL (Figure 8 (d)). Comparing AINiCo with NetSTO, we have two observations:

First, AINiCo brings performance improvement for these 4 CC protocols. With 20 workers, AINiCo improves the throughput by 2.45× (OCC), 1.77× (TicToc), 2.45× (MVCC), and 2.28× (2PL), respectively. This is because AINiCo provides a generalized hotness feedback interface and affinity feedback interface to generate the worker states and global states. The feedback mechanism in AINiCo is generalized for various concurrency control protocols.

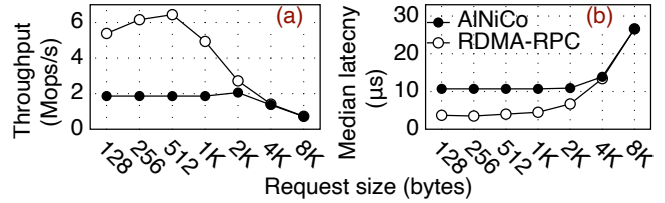


Figure 9: RPC throughput (a) and median latency (b) with varying request sizes.

		L=128	L=256	L=512	L=1K	L=4K
Accuracy rate	TPC-C	22.7%	36.7%	59.8%	85.5%	99.9%
	YCSB-T	5.2%	6.5%	9.9%	28.4%	84.4%
	YCSB-HOT	17.6%	20.7%	29.4%	42.7%	79.8%
Computation cycles		37	37	54	101	N/A
BRAM/LUT/FF(%)		6/2/9	10/3/11	16/5/16	30/10/26	N/A

Table 3: The trade-off of feature vector length L.

Second, 2PL performance is worse than other protocols. This is because 2PL needs to write shared memory to acquire the read lock, while the weakness of OCC's high rollback overhead is negligible in the in-memory system. MVCC has the extra overhead of maintaining version information, so it has lower overall throughput than OCC and TicToc.

5.6 Overhead and Limitation

We evaluate the overhead of AINiCo and discuss its limitation.

Overhead of clients. The client specifies the request feature vector when serializing a transaction into a network message. It takes about 23ns for each key in the transaction parameters.

Overhead of SmartNICs. Figure 9 shows the performance of RDMA-RPC and AINiCo under a micro-benchmark, where the server sends an 8-byte reply to clients immediately as soon as receiving a request. When the request is less than 2KB, the IOPS and latency are limited by the on-NIC scheduler. This is because AINiCo needs extra bandwidth to send the feature vector, and the off-path SmartNIC introduces additional latency.

Overhead of server feedback. The evaluation shows that AINiCo uses only 1.2% of the CPU resources for the software feedback since the FPGA completes most computations for scheduling. However, the CPU-based AINiCo (AINiCo-CPU-N version) takes 27.7% of the CPU resources (i.e., scheduling overhead) to make scheduling decisions, which overshadows the scheduling benefits. This illustrates the need for using SmartNICs to reduce scheduling overhead.

The trade-off of feature vector length L. Table 3 shows the trade-off for choosing the feature vector length L, where the accuracy rate means the proportion of keys detected as the contention that are truly contention. We have two observations. First, a larger feature vector length L improves the accuracy rate because it reduces mapping collisions in features. Second, the scheduler IP core in AINiCo requires more computation cycles and FPGA resources, and they increase linearly with

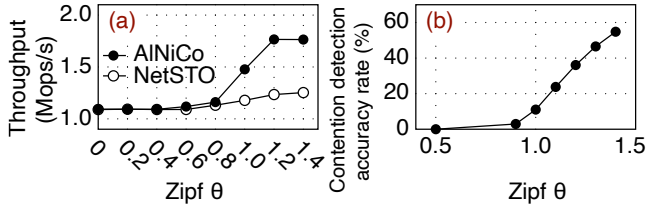


Figure 10: YCSB-T (W:50) throughput (a) and contention detection accuracy rate (b) with varying θ .

the feature vector length L . The theoretical maximum of L is limited by BRAM resources that are used to receive request feature vectors and store scheduler runtime states. We can sacrifice computation optimization or use on-NIC DDR4 (8GB in Innova-2) as storage resources to support a larger L .

Limitation. Based on all experiments in this section, we discuss the following two limitations. 1) AINiCo can not improve the performance of uniform workloads. Figure 10 shows the throughput and the accuracy rate of contention detection with different θ under YCSB-T(W:50%). We observe that only when θ is higher than 0.8, the feature vectors can reflect the contention between requests. This is because, with the more skewed access pattern, the mapping collisions are fewer, and the weights of groups are more distinguishing. 2) AINiCo can not improve the performance of the workloads whose throughput is limited by the NIC bandwidth. We focus on workloads whose throughput is limited by contention because AINiCo consumes additional bandwidth for feature vectors.

6 Related Work

In-network scheduling. There have been intensive evolution efforts in application layer network scheduling, focusing on core affinity, load balance, head-of-line blocking, and in-network transaction coordination.

Core affinity. RSS [25] and FlowDirector [26] dispatch packets based on hashing header fields. MICA [23] uses RSS to assign single-key KV requests to cores based on the key hash partitioning for object-level core affinity. RSS++ [61] achieves dynamic load balance by RSS indirection and supports stateful flow migration by optimizing state transfers among cores. Different from them, AINiCo focuses on contention-aware scheduling for transaction requests.

Load balance. Recent studies [62–64] achieve μ s-scale SLOs through dynamic core scheduling or request scheduling. Humphries et al. [27] offload Shinjuku [64] to SmartNICs. RPCValet [28] and R2P2 [65] dispatch stateless RPC by emulating the theoretically optimal single-queue scheduling policy on NICs and programmable switches, respectively. RackSched [66] is a rack-level service scheduler with two-layer (i.e., inter/intra-server) scheduling.

Head-of-line blocking. In Minos [39] and DARC [40], KV requests for records of different sizes go to different cores to avoid blocking small requests by long-running requests.

In-network transaction coordination. Recent work offloads the transaction coordination to programmable switches [67–69] and client-side SmartNICs [70] to reduce the network overhead of distributed transactions. AINiCo focuses on scheduling single-machine transactions to different CPU cores.

Transaction scheduling. Recent work for transaction processing can be categorized into inter-transaction scheduling and intra-transaction scheduling. AINiCo focuses on inter-transaction scheduling, which schedules each entire transaction to the most appropriate CPU core.

Inter-transaction scheduling. The common principle of batching-based scheduling methods is to make each group almost conflict-free, and they differ in the approaches to residual conflicts. Calvin [71], LADS [19], and QueCC [18] keep track of the dependencies between transaction groups and wait for the completion of dependent transactions. Ding et al. [20] present a method that retries conflicting transactions at a higher priority in the next batch. Further, Jepsen et al. [15, 16] use a programmable switch to triage transactions belonging to different static data partitions before sending them to the database server.

Intra-transaction scheduling. A series of studies use strategies including tracking transaction dependencies [72], exploring the operation commutability [73], reading values from the write buffer [20], and releasing the lock in advance [74] to schedule each read/write operation. Polyjuice [75] uses machine learning models to specify different execution policies for each operation. Moreover, some studies focus on the order of locks; QURO [76] allows transactions that are more likely to block the system to hold locks, while Chiller [77] changes the lock order to reduce the locking time of hot records. Their intra-transaction scheduling is complementary to AINiCo.

7 Conclusion

This paper presents AINiCo, a transaction system that leverages SmartNICs to intelligently schedule incoming transaction requests to CPU cores, minimizing contention with low latency. AINiCo describes the contention in a hardware-friendly manner so that specialized hardware can efficiently make scheduling decisions, and co-designs hardware and software to enable flexible and adaptive scheduling. Evaluation with real hardware (Innova-2) shows that AINiCo reduces the contention between the running transactions and significantly improves performance.

Acknowledgements

We sincerely thank our shepherd and the anonymous reviewers for their valuable feedback, which greatly improved this paper. We also thank Jian Gao, Minhui Xie, Jing Wang, Wenhao Lv, Xiaojian Liao, and Jian Chen for their suggestions. This work is funded by the National Natural Science Foundation of China (Grant No.62022051, 61832011), and Kuaishou.

References

- [1] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment*, 10(2):49–60, 2016.
- [2] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, November 2014.
- [3] Mohammad Sadoghi and Spyros Blanas. Transaction processing on modern hardware. *Synthesis Lectures on Data Management*, 14(2):1–138, 2019.
- [4] Youmin Chen, Xiangyao Yu, Paraschos Koutris, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Plor: General transactions with predictable, low tail latency. In *Proceedings of the 2022 International Conference on Management of Data, SIGMOD/PODS '22*, page 19–33, New York, NY, USA, 2022. Association for Computing Machinery.
- [5] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.
- [6] Standard Specification. TPC BENCHMARK™ C. 1994.
- [7] Tianyang Jiang, Guangyan Zhang, Zhiyue Li, and Weimin Zheng. Aurogon: Taming aborts in all phases for distributed In-Memory transactions. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 217–232, Santa Clara, CA, February 2022. USENIX Association.
- [8] Mellanox. InnoVA™-2 Flex Open Programmable SmartNIC. <https://www.mellanox.com/files/doc-2020/pb-innova-2-flex.pdf>, 2020.
- [9] Cisco. Cisco Nexus SmartNIC. <https://www.cisco.com/c/en/us/products/interfaces-modules/nexus-smartnic/index.html>, 2021.
- [10] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.
- [11] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-oriented transaction execution. *Proceedings of the VLDB Endowment*, 3(ARTICLE), 2010.
- [12] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. Schism: a workload-driven approach to database replication and partitioning. 2010.
- [13] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 61–72, 2012.
- [14] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. Sword: Scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology, EDBT '13*, page 430–441, New York, NY, USA, 2013. Association for Computing Machinery.
- [15] Theo Jepsen, Alberto Lerner, Fernando Pedone, Robert Soulé, and Philippe Cudré-Mauroux. In-network support for transaction triaging. 2021.
- [16] Theo Jepsen. *Building blocks for leveraging in-network computing*. PhD thesis, Università della Svizzera italiana, 2020.
- [17] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. Q-store: Distributed, multi-partition transactions via queue-oriented execution and communication. In *EDBT*, pages 73–84, 2020.
- [18] Thamir M Qadah and Mohammad Sadoghi. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*, pages 13–25, 2018.
- [19] Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Transactions on Knowledge and Data Engineering*, 28(10):2635–2650, 2016.
- [20] Bailu Ding, Lucja Kot, and Johannes Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment*, 12(2):169–182, 2018.
- [21] Guna Prasaad, Alvin Cheung, and Dan Suciu. Handling highly contended oltp workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 527–542, 2020.

- [22] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for optimism in contended main-memory multicore transactions. *Proceedings of the VLDB Endowment*, 13(5):629–642, 2020.
- [23] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.
- [24] Antoine Kaufmann, Simon Peter, Thomas Anderson, and Arvind Krishnamurthy. Flexnic: Rethinking network dma. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems, HOTOS’15*, page 7, USA, 2015. USENIX Association.
- [25] Intel. Receive-Side Scaling (RSS). <http://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf>, 2007.
- [26] Intel. Ethernet Flow Director. <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf>, 2014.
- [27] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. Mind the gap: A case for informed request scheduling at the nic. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 60–68, 2019.
- [28] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. Rpcvalet: Ni-driven tail-aware balancing of μ s-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–48, 2019.
- [29] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. Elastic rss: Co-scheduling packets and cores using programmable nics. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, pages 71–77, 2019.
- [30] Intel. Infrastructure Processing Units (IPUs). <https://www.intel.com/content/www/us/en/products/network-io/smartnic.html>, 2021.
- [31] Jiaxin Lin, Kiran Patel, Brent E Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259, 2020.
- [32] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 44–57, 2016.
- [33] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-efficient microservices on smartnic-accelerated servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, 2019.
- [34] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 137–152, 2017.
- [35] Nikita Lazarev, Neil Adit, Shaojie Xiang, Zhiru Zhang, and Christina Delimitrou. Dagger: Towards efficient rpcs in cloud microservices with near-memory reconfigurable nics. *IEEE Computer Architecture Letters*, 19(2):134–138, 2020.
- [36] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafir. Autonomous nic offloads. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021*, page 18–35, New York, NY, USA, 2021. Association for Computing Machinery.
- [37] Zsolt István. Let’s add transactions to fpga-based key-value stores! In *Proceedings of the 16th International Workshop on Data Management on New Hardware, DaMoN ’20*, New York, NY, USA, 2020. Association for Computing Machinery.
- [38] Zhaoshi Li, Leibo Liu, Yangdong Deng, Jiawei Wang, Zhiwei Liu, Shouyi Yin, and Shaojun Wei. Fpga-accelerated optimistic concurrency control for transactional memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO ’52*, page 911–923, New York, NY, USA, 2019. Association for Computing Machinery.
- [39] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 79–94, 2019.
- [40] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing

- tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21*, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery.
- [41] Yihe Huang, Nathaniel Herman, William Qian, Jeevana Priya Inala, Eddie Kohler, Lillian Tsai, Barbara Liskov, and Liuba Shrira. STO: Software Transactional Objects. <https://github.com/readablesystems/sto/>, 2021.
- [42] Mellanox. How To Implement PeerDirect Client using MLNX_OFED. [PeerDirect](#), 2018.
- [43] PCI-SIG. PCI-Express Specification. <https://www.pcisig.com/specifications/pciexpress/>, [n. d.].
- [44] Wojciech M Zabołotny. Dma implementations for fpga-based data acquisition systems. In *Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2017*, volume 10445, pages 1269–1276. SPIE, 2017.
- [45] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.
- [46] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1629–1642, 2016.
- [47] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multicore in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 21–35, 2017.
- [48] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. *ACM sigplan notices*, 44(6):155–165, 2009.
- [49] Dixin Tang, Hao Jiang, and Aaron J Elmore. Adaptive concurrency control: Despite the looking glass, one concurrency control does not fit all. In *CIDR*, volume 2, page 1. Citeseer, 2017.
- [50] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.
- [51] Masoud Hemmatpour, Bartolomeo Montrucchio, Maurizio Rebaudengo, and Mohammad Sadoghi. Analyzing in-memory nosql landscape. *IEEE Transactions on Knowledge and Data Engineering*, 34(4):1628–1643, 2022.
- [52] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: An rdma-enabled distributed persistent memory file system. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 773–785, USA, 2017. USENIX Association.
- [53] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, 2016.
- [54] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, page 437–450, USA, 2016. USENIX Association.
- [55] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable rdma rpc on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.
- [56] Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. Th-dpms: Design and implementation of an rdma-enabled distributed persistent memory storage system. *ACM Trans. Storage*, 16(4), oct 2020.
- [57] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with In-Network cache coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 277–292. USENIX Association, February 2021.
- [58] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD/PODS '22, page 1033–1048, New York, NY, USA, 2022. Association for Computing Machinery.
- [59] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

- [60] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. *Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook*, page 209–224. USENIX Association, USA, 2020.
- [61] Tom Barbette, Georgios P Katsikas, Gerald Q Maguire Jr, and Dejan Kostić. RSS++ load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 318–333, 2019.
- [62] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.
- [63] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.
- [64] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μ second-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.
- [65] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*, pages 863–880, 2019.
- [66] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240, 2020.
- [67] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation, NSDI’16*, page 425–438, USA, 2016. USENIX Association.
- [68] Theo Jepsen, Leandro Pacheco de Sousa, Masoud Moshref, Fernando Pedone, and Robert Soulé. Infinite resources for optimistic concurrency control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing, NetCompute ’18*, page 26–32, New York, NY, USA, 2018. Association for Computing Machinery.
- [69] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP ’17*, page 104–120, New York, NY, USA, 2017. Association for Computing Machinery.
- [70] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP ’21*, page 740–755, New York, NY, USA, 2021. Association for Computing Machinery.
- [71] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.
- [72] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, 2014.
- [73] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase reconciliation for contended in-memory transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 511–524, 2014.
- [74] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking (extended version). *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, 2021.
- [75] Jiachen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. Polyjuice: High-performance transactions via learned concurrency control. *arXiv preprint arXiv:2105.10329*, 2021.
- [76] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. Contention-aware lock scheduling for transactional databases. *Proceedings of the VLDB Endowment*, 11(5):648–662, 2018.
- [77] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 511–526, 2020.



FpgaNIC: An FPGA-based Versatile 100Gb SmartNIC for GPUs

Zeke Wang¹, Hongjing Huang¹, Jie Zhang¹, Fei Wu^{1,2}

¹ Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, China

² Shanghai Institute for Advanced Study of Zhejiang University, China

Gustavo Alonso

Systems Group, Dept. of Computer Science

ETH Zurich, Switzerland

Abstract

Network bandwidth is improving faster than the compute capacity of the host CPU, turning the CPU into a bottleneck. As a result, SmartNICs are often used to offload packet processing, even application logic, away from the CPU. However, today many applications such as Artificial Intelligence (AI) and High Performance Computing (HPC) rely on clusters of GPUs for computation. In such clusters, the majority of the network traffic is created by the GPUs. Unfortunately, commercially available multi-core SmartNICs, such as BlueField-2, fail to process 100Gb network traffic at line-rate with its embedded CPU, which is capable of doing control-plane management only. Commercially available FPGA-based SmartNICs are mainly optimized for network applications running on the host CPU. To address such scenarios, in this paper we present FpgaNIC, a GPU-oriented SmartNIC to accelerate applications running on distributed GPUs. FpgaNIC is an FPGA-based, GPU-centric, versatile SmartNIC that enables direct PCIe P2P communication with local GPUs using GPU virtual address, and that provides reliable 100Gb network access to remote GPUs. FpgaNIC allows to offload various complex compute tasks to a customized *data-path accelerator* for line-rate in-network computing on the FPGA, thereby complementing the processing at the GPU. The data-path accelerator can be programmed using C++-based HLS (High Level Synthesis), so as to make it easier to use for software programmers. FpgaNIC has been designed to explore the design space of SmartNICs, e.g., direct, on-path, and off-path models, benefiting different type of application. It opens up a wealth of research opportunities, e.g., accelerating a broad range of distributed applications by combining GPUs and FPGAs and exploring a larger design space of SmartNICs by making them easily accessible from local GPUs.

1 Introduction

While the computing capacity of CPUs is growing slowly and mostly either through parallelism (SIMD, multi-core) or specialization (GPGPU, security or virtualization support), network bandwidth is growing obviously faster. 100Gbps

NICs are common and soon 400Gbps will be available [46]. This growing gap between network bandwidth and compute capability is being addressed through offloading of network functions to the Network Interface Card (NIC), so called SmartNIC [13, 14, 19, 36, 45], which frees up significant CPU cycles and provides better hardware to keep up with the growing network traffic and its often strict requirements in terms of bandwidth and latency.

Modern GPUs provide an order of magnitude higher memory bandwidth and higher compute capacity than modern CPUs. As a result, GPUs have become a key element in, e.g., Artificial Intelligence (AI) and High Performance Computing (HPC) applications that are both compute- and memory-bound [4]. Since a multi-GPU server is often not enough to cover the computing power needed in many AI, graph, and HPC applications, current solutions are typically based on a cluster of GPUs (e.g., [27, 58, 78]), with the GPUS generating the majority of the network traffic in such systems.

In this paper, we present the design of a 100Gb GPU-centric SmartNIC to serve distributed applications running on GPUs. From a GPU's perspective, such a SmartNIC should 1) enable the GPU directly triggering doorbell registers and polling on status registers on the SmartNIC without CPU intervention (**G1**); 2) use the GPU virtual address space to directly access GPU memory via Peer-to-Peer (P2P) communication without CPU intervention (**G2**); 3) implement in hardware the full network stack to ensure low latency and high throughput (**G3**); 4) support application logic offloading to a software-defined and hardware-accelerated data-path accelerator, i.e., *on-NIC* computing processing 100Gb network traffic at line-rate (**G4**)¹; and 5) The data-path accelerator should be easily programmed by system programmers (**G5**). Commercially available SmartNICs are not able to satisfy all these goals as they are not optimized for GPUs. In the following, we analyze existing multicore and FPGA-augmented SmartNICs that motivate FpgaNIC.

Multicore SmartNIC. A multicore SmartNIC, such as

¹In the paper, we use on-NIC computing module and data-path accelerator interchangeably.

Table 1: Comparison of FpgaNIC with existing SmartNIC types for GPUs. ✓ indicates full support, ✗ indicates no support, and ✓ indicates partial support.

	Multicore SmartNIC [44]	FPGA-aug. SmartNIC [45]	Ours
Control plane offload (G1)	✗	✗	✓
Access GPU with virtual address (G2)	✓	✗	✓
100Gb transport offload (G3)	✓	✓	✓
100Gb data-path accelerator (G4)	✗	✓	✓
High programmability (G5)	✓	✗	✓

BlueField-2 [44], combines a multicore CPU, e.g., ARM, with an ASIC network controller. It introduces an additional hop to implement the smart function using a multicore CPU, which features two DDR4 channels for staging. This allows to map a broad range of applications on multicore SmartNICs. Therefore, its high programmability **G5** is fully supported. However, it increases processing latency and multicore CPU’s memory bandwidth can easily become a performance bottleneck. BlueField-2 has 27.3GB/s achievable memory bandwidth under a benchmarking tool sysbench [1], indicating that directly staging 100Gbps data stream at the NIC CPU already overwhelms BlueField-2, matching the findings in [40]. Therefore, it cannot act as a 100Gb data-path accelerator **G4**. To our knowledge, the multicore SmartNIC is controlled from the host CPU, so **G1** is not yet supported. The network transport is implemented with the packet processing engine with necessary control on the host (or ARM) CPU, so **G3** is partially supported. Finally, the ASIC network chip of multicore SmartNIC supports NVIDIA GPUDirect [52], which enables direct PCIe P2P data communication to a GPU, so **G2** is fully supported.

FPGA-augmented SmartNIC. An FPGA-augmented SmartNIC combines a hardware-programmable FPGA with an ASIC network controller. For example, Mellanox Innova-2 [45] is an FPGA-augmented SmartNIC featuring a network adapter ConnectX-5 and an Xilinx FPGA. ConnectX-5 consists of a 100Gbps InfiniBand/Ethernet interface for networking and a PCIe Gen4x8 interface for communicating with the host CPU. The FPGA communicates with ConnectX-5 via a PCIe x8 Gen4 link, so processing packets on the FPGA adds considerable latency to the packets and processing cannot happen at line rate because Innova-2 has limited PCIe link bandwidth between the FPGA and ConnectX-5. Therefore, Innova-2 can only acts as a partial 100Gb data-path accelerator **G4**. **G1** is not yet supported, **G2** is not supported, **G3** is partially supported, and the high programmability **G5** is not supported.

Given the limitations of existing NICs, in this paper we present FpgaNIC, a full-stack FPGA-based GPU-centric versatile SmartNIC that opens up the opportunity to explore a *large design space around SmartNICs* due to the FPGA’s reconfigurable nature and efficient *FPGA-GPU co-processing* while achieving all the five goals mentioned above in a single

system. We have implemented FpgaNIC as a composable architecture that consists of a *GPU communication stack*, a *100Gb hardware network transport*, and an *On-NIC computing (ONC)*, i.e., data-path accelerator.² The GPU communication stack enables offloading of control plane onto GPUs (**G1**) and thus *for the first time* enables local GPUs directly to manipulate SmartNIC without CPU intervention, and enabling the FPGA-based SmartNIC *for the first time* to use GPU virtual address to directly access GPU memory via PCIe P2P communication (**G2**). The 100Gb hardware network transport enables efficient and reliable 100Gb network communication with remote GPUs (**G3**). Moreover, FpgaNIC adopts a layered design to allow developers to easily explore the design space of SmartNIC models (i.e., direct, off-path, and on-path) to benefit their application, where different applications favor a different SmartNIC model. FpgaNIC allows to prototype applications that can eventually be migrated to hardened SmartNICs. Implementing a data-path accelerator on an FPGA can easily satisfy line-rate processing requirement (**G4**) due to its hardware implementation, while FpgaNIC allows to use C++-based High Level Synthesis (HLS) so as to provide high programmability (**G5**). As such, in the context of FPGA-GPU co-processing, the GPU provides to applications expressiveness and computing flexibility, while the FPGA provides a flexible network infrastructure and the necessary ONC. FpgaNIC results in significant end-to-end performance improvements as data can be processed as it flows from/to the GPU in a streaming manner and without involving the CPU.

We have prototyped FpgaNIC on a PCIe-based Xilinx FPGA board Alveo U50 [74], whose UltraScale+ FPGA features a 100Gbps networking port, a X16 PCIe Gen3, and 8GB HBM. Its form factor is half-length, half-height and its Maximum Total Power (MTP) is 75W, allowing it to be easily deployed in any CPU server.³ In addition to comprehensive benchmarking, we validate the **versatility** and **potential** of FpgaNIC by implementing use cases for all three models: GPU-centric networking (in a direct model), a collective primitive AllReduce (in an off-path model), and cardinality estimation on incoming streaming data (in an on-path model). The experimental results show that FpgaNIC is able to efficiently support all three SmartNIC models at the full line rate of 100 Gbps Ethernet. Particularly, FpgaNIC-enhanced AllReduce almost reaches the maximum theoretical throughput when performing on a distributed pool of eight RTX 8000 GPUs, while requiring fewer than 20% of the FPGA resources on the U50 board. It indicates that, even when considering the full network stack offloading, it has sufficient FPGA resources to allow more aggressive offloading, e.g., the Adam

²In the paper, we use on-NIC computing module and data-path accelerator interchangeably.

³We have also migrated FpgaNIC onto the Alveo U280 FPGA board [73] with minor modifications affecting the FPGA pin mapping. Though we have not ported FpgaNIC to Intel FPGA boards yet, we believe that it requires only a small amount of effort to do so. We leave the porting to future work.

2.2 Main Architecture of FpgaNIC

To address the above four challenges, FpgaNIC adopts a layered design to enable easy design space exploration for SmartNIC architectures dedicated for various distributed applications that run on distributed GPUs, while minimizing the development effort and increasing the overall system efficiency. FpgaNIC consists of three main components: *GPU communication stack*, *reliable network transport in hardware*, and *on-NIC computing* (ONC), as shown in Figure 1. The goal of the GPU communication stack is 1) to allow the FPGA to use GPU virtual address (C1) to directly access GPU memory via direct PCIe P2P data communication at low-latency and line-rate, and 2) to allow GPUs to initiate data transfers by using *doorbell registers* on the FPGA to avoid having to involve the host CPU in the invocation. The goal of reliable network transport in hardware is to provide a reliable, low-latency, and high-throughput network access to the local GPUs (C2). The goal of on-NIC computing is 1) to enable high-level programming interface, and 2) to enable three NIC models: direct, on-path, and off-path, such that FpgaNIC is able to benefit a broad range of GPU-powered distributed applications (C4).

2.3 GPU Communication Stack

Built on a PCIe IP core, e.g., Xilinx’s UltraScale+ Gen3 x16, the GPU communication stack of FpgaNIC aims at enabling offloading the control plane onto GPUs (via a slave interface) and offloading the data plane onto the FPGA (via a master interface), such that the host CPU is bypassed.

2.3.1 Offloading Control Plane onto GPUs

In order to allow GPUs to directly access the FPGA’s control and status registers, FpgaNIC needs to offload the control plane onto the GPUs.

How to Enable Control Plane Offloading? Enabling control plane offloading requires a hardware-software codesign approach. On the hardware side, we enable a PCIe BAR exposing a configurable FPGA address space at the PCIe IP core on the FPGA. On the software side, our implementation consists of a *GPU driver*, an *FPGA driver*, and *user code* that interacts with both drivers. The process consists of three steps. First, the FPGA driver uses the function *misc_register* to register the PCIe BAR with the Linux kernel as an IO device */dev/fpga_control*. Second, the user code uses the function *mmap* to map the device into the host address. Third, the user code adopts a CUDA (Compute Unified Device Architecture) memory management function to register the host address for use within a CUDA kernel [52]. With this, the GPU can directly trigger doorbell registers and poll status registers on the FPGA without CPU intervention.

What Control Plane Offloading can Do? After enabling control plane offloading, the doorbell/status registers that are instantiated by all the components (GPU communications stack, ONC, and network transport) have to be mapped into

GPU virtual address space so that the GPU program is able to access these registers without CPU intervention. Moreover, it enables us to populate GPU TLB (GTLB) entries on the FPGA such that the FPGA can translate GPU virtual address to physical address before issuing a DMA read/write operation to GPU memory (§2.3.2).

2.3.2 Offloading Data Plane onto the FPGA

FpgaNIC needs to offload the data plane onto the FPGA to allow the FPGA to directly access the GPU memory. However, NVIDIA GPUDirect [52] allows direct PCIe P2P data communication using physical address. For the sake of easy programming, FpgaNIC needs to work on GPU virtual address, rather than physical address (C1). Via a GPU BAR window, Tesla GPUs expose all of their device memory space, e.g., 40GB, while Quadro GPUs typically expose 256MB memory space with 36MB reserved for internal use [52]. In order to allow the FPGA to access more GPU memory space, FpgaNIC needs to store all the related virtual to physical address translation entries. To minimize the overhead of translation, we intend to keep all the entries on on-chip memory. However, the 64KB GPU page size becomes the main challenge, because storing a great number of translation entries on the FPGA needs a large on-chip memory. For example, 32GB GPU memory needs 512K entries, far beyond the number the FPGA implementation can accommodate without hurting timing.

How to Enable FPGA to Efficiently Work on Virtual Address? To this end, we propose a GPU Translation Lookaside Buffer (GTLB) to perform address translation on the FPGA, while keeping the on-chip memory consumption reasonably low. The key motivation behind the design of GTLB is that even though a single contiguous virtual address space needs not be physically contiguous on GPU memory, it has high probability to be physically contiguous, especially at the granularity of 2MB. Therefore, we manually coalesce 32 consecutive 64KB GPU memory pages into a 2MB page if these 64KB pages are allocated to a contiguous portion of physical memory and aligned within the 2MB page. The GTLB consists of a *main TLB* and a *complementary TLB*. The process of populating the GTLB on the FPGA involves four steps, as shown in Algorithm 1. First, we pre-alloc GPU memory space using *gpuMemAlloc* for staging the GPU memory that will be accessed by the DMA engines on the FPGA (Line 1). Second, we pass the initial virtual address and length of this GPU memory to the GPU kernel function *nvidia_p2p_put_pages* to get all the $\langle VA, PA \rangle$ pairs for all the 64KB pages (Line 2), where VA refers to virtual address and PA refers to physical address. Third, we try to coalesce 64KB pages into 2M pages as aggressive as possible (Line 3). Fourth, we populate main and complementary TLBs (Lines 4-18) via the control registers exposed by the control plane offloading (§2.3.1). The main TLB provides the virtual to physical address translations for 2MB pages (Lines 7-10). If any 2MB page is not physi-

Algorithm 1: POPULATING GTLB

```
Input   : init_addr: initial GPU virtual address
           len: length of GPU memory
Output  :  $TLB^{main}$ : main TLB
            $TLB^{comp}$ : complementary TLB
/* Step 1: Malloc GPU memory space. */
1 init_addr = gpuMemAlloc(len);
/* Step 2: Get <VA, PA> pairs of all the 64KB pages. */
2 <VA64KB, PA64KB> pairs = nvidia_p2p_put_pages(init_addr, len);
/* Step 3: Coalescing 64KB pages to 2MB pages if possible */
3 <VA2MB, PA2MB> pairs ← <VA64KB, PA64KB> pairs;
/* Step 4: Populating  $TLB^{main}$  and  $TLB^{comp}$  */
4 index = 0; /* Large page index */
5 comp = 0; /* Base page index */
6 for (pair in <VA2MB, PA2MB> pairs) do
7   if (pair is physically contiguous) then
8     /* Update the  $TLB^{main}$  */
9      $TLB^{main}[index].pair$  = pair;
10     $TLB^{main}[index].valid$  = 1;
11   end
12   else
13     /* Update the  $TLB^{comp}$  */
14      $TLB^{comp}[32 * comp + 31 : 32 * comp]$  = pair's 32 64KB pages;
15      $TLB^{main}[index].valid$  = 0;
16      $TLB^{main}[index].comp\_offset$  = comp*32;
17     comp++;
18   end
19 index++;
end
```

cally contiguous, we store the corresponding 32 translations of 32 64KB pages in the complementary TLB, which provides 2048 entires for accommodating 64 such 2M pages (Lines 12-15).⁵ As such, the total number of required entires for 32GB memory becomes 16K+2K=18K, significantly smaller than the previous 512K entries.

Fully-pipelined Translation Lookup. After the population, FpgaNIC is able to directly access GPU memory using on-line virtual to physical address translation. Given a virtual address, FpgaNIC first checks the corresponding entry in the main TLB to see whether it is continuous or not ($TLB^{main}.valid == 1$). If yes, FpgaNIC fetches the PA and feeds it into the DMA engine. If no, FpgaNIC will read the corresponding entry in the complementary TLB using the offset $TLB^{main}.comp_offset$. We can observe that the proposed GTLB can easily achieve fully-pipelined translation lookup on the FPGA.

GTLB Miss/Eviction. Currently, we pre-populate TLB entries for each application, assuming that the FPGA only accesses certain range of GPU memory. When GTLB miss or eviction happens, we need to re-populate GTLB entries for successful GPU memory references. However, we suggest to instantiate multiple GTLBs to provide sufficient number of GTLB entries to eliminate potential GTLB misses and evictions on the FPGA, because each GTLB entry only occupies a row of a BRAM, indicating relatively low cost of storing GTLB entries on the FPGA.

⁵In our experiment, 2048 entires are far beyond enough.

2.4 100Gbps Hardware Network Transport

In order to address the second challenge (C2), FpgaNIC offloads the transport-layer network to the FPGA to provide a reliable and high-performance hardware network transport to the local GPUs. Fortunately, there is a growing amount of open-source FPGA-based 100Gb network stacks such as the TCP/IP stack of [57, 60] and the RoCEv2 stack used in [62]. Without loss of generality, FpgaNIC is built on the 100Gb TCP/IP stack [57, 60], which is able to support thousands of connections with external FPGA memory for buffering.⁶ We have modified this stack to adapt it to the requirements of FpgaNIC's by modifying its interface to improve bandwidth utilization and allow local GPUs to directly control the network transport.

2.4.1 Efficient Decoupled Application Interface

The original application interface [25, 57, 60] requires a control handshake between the TCP stack and the application code before sending or receiving a network packet to or from the TCP stack. A control handshake takes from 10 to 30 cycles while the payload of a packet (up to 1460B) only takes up to 23 cycles, leading to low network bandwidth utilization. To reduce the overhead of the handshake, we introduce an efficient decoupled application interface that does not need the handshake and further overlaps the control handshake and the packet transfer, maximizing the network bandwidth utilization and easing programming.

Decoupled Sending Application Interface. The original sending interface only allows to send a data chunk at a time after a control handshake, where the size of a data chunk is up to 1460B. A data chunk and a TCP header constitute a TCP segment, which can be encapsulated into an IP packet before sending over Ethernet. The proposed decoupled interface gets rid of the handshake, overlapping the control handshakes with the data transfer. And it further allows to send data streams of up to 4GB in size by automatically splitting the data stream into the right size chunks without programmer's involvement in packetization.

Decoupled Receiving Application Interface. The original receiving interface informs the user logic through a valid notification when a TCP segment is available to be consumed, which then sends out the read request to the receiving interface. After 10 to 30 cycles, the TCP segment's payload will be available at the 64B-wide AXI (Advanced eXtensible Interface)-Stream interface and consumed by the user logic. Similar to the sending interface, the proposed decoupled interface gets rid of the handshake, and further overlaps handshake

⁶The TCP stack needs two 64KB fixed-sized buffers per connection, one buffer for incoming packets and the other for outgoing packets. Therefore, external FPGA memory is needed to support thousands of concurrent connections. However, if fewer than 10 concurrent connections are needed, the TCP stack of [57, 60] can implement the buffers using on-chip memory such that external FPGA memory could be saved for offloaded smart functionality. In this paper, FpgaNIC uses both versions.

Table 2: Resource Usage breakdown of FpgaNIC on U50.

	LUTs	REGs	RAMs	DSPs
Available	871K	1743K	232.4Mb	9024
GPU Commu. Stack	79K	103K	5.2Mb	0
100G HW Transport	101.3K	166.5K	23.4Mb	0
ONC: GPU-centric networking	14.5K	20K	24.6Mb	0
ONC: AllReduce	7.3K	10K	12.8Mb	0
ONC: Hyperloglog	19.5K	26K	7.1Mb	1104

and data transfer and assembles the complete data stream for each TCP connection without programmer’s involvement in depacketization.

2.5 On-NIC Computing (ONC)

The on-NIC computing module sits between the GPU communication stack module and the 100Gbps network hardware transport module, so ONC can directly manipulate the other two modules to enable flexible design space exploration around GPU-centric SmartNICs. The key goal of on-NIC computing module is to 1) expose high-level programming interface for system programmer, and 2) enable three SmartNIC models for various GPU-powered distributed applications. In the following, we discuss the programming interface of ONC and how to enable three three models.

2.5.1 High-level Manipulation Interfaces of ONC

In order to address the third challenge (C3), FpgaNIC intends to raises the programming abstraction from HDL to high-level synthesis (HLS), i.e., C/C++, such that systems programmers are able to use C/C++ to manipulate FpgaNIC, rather than cycle-sensitive HDL.⁷ In the following, we present the concrete manipulation interfaces for the GPU communication stack and hardware network transport modules.

Manipulation Interfaces of GPU Communication Stack. The GPU communication stack exposes two manipulation interfaces: a slave interface that allows GPUs to access FPGA’s registers and a master interface that allows the FPGA to directly access GPU memory, as shown in Table 3.

The slave interface is a 4B-wide AXI-Lite interface (*axilite_control*), through which local GPUs directly access doorbell and status registers within FpgaNIC without CPU intervention. In FpgaNIC, we instantiate 512 doorbell registers and 512 status registers, each of which has its own PCIe address to allow individual access. We correspond a few doorbell and status registers to each *engine* from any of three components within FpgaNIC. The doorbell registers can be triggered by GPUs to manipulate the engine, and the status registers can be polled by GPUs to check the status of the engine.

The master interface consists of two command streams and two data streams. The two command streams are 96-bit-wide AXI-Stream interfaces (*dma_read_cmd* and *dma_write_cmd*) that provide the GPU virtual address and length to directly access GPU memory, where the length is

⁷Nevertheless, ONC can be also programmed in HDL if necessary.

Table 3: Two interfaces of GPU communication stack

Type	Interface	Content
Slave interface	<i>axilite_control</i>	AXI-Lite interface for configuration
Master interface	<i>dma_read_cmd</i>	Dest. GPU virtual address, length
	<i>dma_read_data</i>	AXI data stream from GPU memory
	<i>dma_write_cmd</i>	Source GPU virtual address, length
	<i>dma_write_data</i>	AXI data stream to GPU memory

Table 4: Manipulation interface for the network transport

Type	Interface	Meaning
Data interface	<i>tcp_tx_meta</i>	Session ID, length
	<i>tcp_tx_data</i>	AXI data stream to remote node
	<i>tcp_rx_meta</i>	Session ID, length, IP, port, etc.
	<i>tcp_rx_data</i>	AXI data stream from remote node
Control interface	<i>server_listen_port</i>	A TCP listening port
	<i>server_listen_start</i>	Starting to listen
	<i>client_conn_port</i>	Destination port to connect
	<i>client_conn_ip</i>	Destination ip to connect
	<i>client_conn_start</i>	Start to connect to server
	<i>conn_close_session</i>	Destination session to connect
	<i>conn_close_start</i>	Start to close connection

up to 4G. The data from and to GPU memory is sent over the *dma_read_data* and *dma_write_data* data streams, which are 64B-wide AXI-Stream interfaces. For either GPU memory read or write operation, we need to configure the command stream and then work on the corresponding data stream, allowing programmers to easily access GPU memory.

Interfaces of Hardware Network Transport. The hardware network transport exposes two interfaces: *data interface* and *control interface*.

The data interface consists of a *sending interface* and a *receiving interface*. The sending interface consists of a metadata stream and a data stream. The metadata stream (*tcp_tx_meta*) is a 48-bit-wide AXI-Stream that provides a 4B-wide data length and a 2B-wide session ID that corresponds to a remote node. The data stream (*tcp_tx_data*) is a 64B-wide AXI-Stream to send payload stream. The receiving interface also consists of a metadata stream and a data stream. The metadata stream (*tcp_rx_meta*) is an 44B-wide AXI-Stream that provides session ID, length, IP address, port and close session flag. The data stream (*tcp_rx_data*) is a 64B-wide AXI-Stream to receive payload stream from remote node.

The control interface of the hardware transport is similar to that of the well-understood socket interface, which allows GPU programmers to easily leverage the network transport, with their meanings as shown in Table 4. We instantiate the corresponding doorbell and status registers, exposed through the PCIe’s slave interface (§2.3), to allow local GPUs to directly manipulate or poll the 100Gb hardware network transport.⁸ In summary, the network transport serves as a network proxy, through which the ONC module and local GPUs can access the network transport directly without CPU intervention, so as to address the second challenge C2.

⁸Inside the FPGA, the ONC module (§2.5) can also directly manipulate the hardware transport via these registers.

Table 5: Lines of code for each component of FpgaNIC

	Hardware	Software
GPU Commu. Stack	2.9K (Verilog/HLS)	0.7K (C++, CUDA)
100G HW Transport	15.3K (HLS)	
ONC: GPU-centric networking	1.0K (HLS)	0.5K (C++, CUDA)
ONC: AllReduce	2.7K (HLS/Verilog)	1.5K (C++)
ONC: Hyperloglog	1.6K (HLS)	0.3K (C++, CUDA)

2.5.2 How to Support Three SmartNIC Models?

In order to address the fourth challenge **C4**, FpgaNIC’s on-NIC computing component allows system programmers to customize data-path engines between the GPU communication stack and the hardware network transport to accelerate various distributed applications. Table 2 shows that the previous GPU communication stack and network transport consume less than 20% FPGA resources on a mid-sized FPGA U50, so the on-NIC computing component has plenty of resources to realize complex data-path engines to accelerate various distributed applications. Moreover, the commercial FPGA board that features DDR4 (even HBM) is able to stage data from network or GPUs, and perform on-NIC computing on the data before feeding into GPUs or sending out to network.

Due to the reconfigurable nature of the FPGA, FpgaNIC can easily support various SmartNIC models: direct, on-path, and off-path, to benefit a broad range of distributed applications. Table 5 shows the lines of code for each component.

The direct model directly exposes the hardware network transport module to local GPUs via the GPU communication stack module, such that local GPUs can directly manipulate the network transport to do reliable network communication. An example, we develop a GPU-centric networking to demonstrate the potentials of the direct model. Due to space limitation, we describe the detailed design and implementation of GPU-centric networking to the Appendix §A.1.

The on-path model is similar to the direct model that local GPUs directly manipulate the hardware network transport, except that the on-path model allows the network stream also to enter an on-path engine in the ONC component for the offloaded computation, where the on-path engine needs to consume the network stream at line-rate such that the on-path engine would not impede line-rate network traffic. We use the HyperLogLog (HLL) application [18, 33] as an example to demonstrate the power of the on-path model. The detailed design and implementation of HLL with FpgaNIC can be found in the Appendix §A.3.

The off-path model enables an off-path engine in the ONC component to directly manipulate the GPU communication stack and the hardware network transport such that FpgaNIC is able to orchestrate the data flow between all the three components. Typically, the off-path needs to stage data in on-board memory. We use the collective communication primitive AllReduce [4, 11, 50] as an example to demonstrate the power of the off-path model (§A.2). The detailed design and

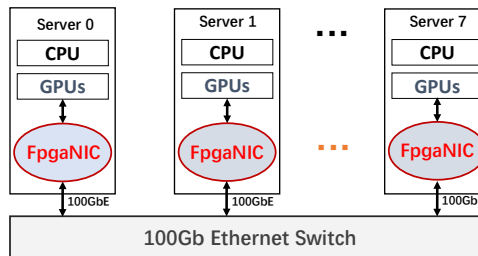


Figure 2: Experimental Setup

implementation of AllReduce with FpgaNIC is in the Appendix §A.2.⁹

How to Support Multiple Tenants? To support multiple tenants, we can adopt Coyote [32] to wire the GPU communication stack and hardware network stack into the static region of FpgaNIC while exposing the same programming interface to offloaded tasks, for which we pre-synthesize the FPGA bitstreams ahead of time. Furthermore, FpgaNIC adopts the notion of *vFPGAs* (virtual FPGAs or separate application regions that are individually reconfigurable) as implemented in Coyote [32] to smoothly support secure, temporal and spatial multiplexing of GPU communication stack and hardware network transport between tenants (without pre-emption and context switching). For each tenant, FpgaNIC provides sufficient FPGA resources in a partial reconfiguration region to implement an independent ONC engine to guarantee performance isolation, and thus we no longer need to reboot the FPGA to change the functionality of FpgaNIC. We leave this as future work.

3 Experimental Evaluation

3.1 Experimental Setup

System Architecture. The experiments are run on a cluster consisting of eight 4U AMAX servers, connected with a Mellanox 100Gbps Ethernet SN2700 switch (Figure 2). Each server is equipped with two Intel Xeon Silver 4214 CPUs @2.20GHz, 128GB memory, FpgaNIC (i.e., a Xilinx Ultra-Scale+ FPGA [72]), and a Nvidia RTX 8000 GPU, where the FPGA and the GPU have direct PCIe P2P communication, as shown in Figure 1. Two servers have an additional two A100 GPUs. FpgaNIC is implemented on Xilinx Alveo cards U50 or U280 with Vivado 2020.1.

Methodology. We first benchmark the GPU communication stack and hardware network transport to demonstrate that FpgaNIC allows easy PCIe P2P communication with local GPUs and reliable network communication with remote GPUs. We then evaluate the three FpgaNIC models: direct (§3.3), off-path (§3.4), and on-path (§3.5), to demonstrate FpgaNIC’s performance and ability to enable the exploration of a large SmartNIC design space.

⁹The off-path model is generic enough such that it would also work well in other applications that follow a partition/aggregate pattern and require multiple rounds of communication [38].

3.2 Benchmarking Shared Infrastructure

We benchmark the shared GPU communication stack and hardware network transport.

3.2.1 GPU Communication Stack

To analyze the effect of control plane and data plane offloading, we measure the latency and throughput of the PCIe P2P link (§2.3). We use two classes of GPU: Quadro RTX8000 (labelled “R8K”) and Tesla A100 (labelled “A100”), since a different GPU class leads to different latency and throughput.

Effect of Control Plane Offloading. We examine the effect of control plane offloading by comparing the latency of commands issued from the GPU to the FPGA. Figure 3 shows the read latency when using various end points: “X_Y” means that device “X” reads from device “Y”. A first important result is that the latency of interactions between the GPU and the FPGA is comparable to that of the CPU calling the FPGA and it is under 1 microsecond. Moreover, the GPU-FPGA’s latency fluctuation is smaller than that of CPU-FPGA, demonstrating one of the advantages of FpgaNIC in terms of offering deterministic latency. The results also show that performance improves slightly with a better GPU, indicating that the overall system will improve with future versions of the GPU. Finally, the latency of “GPU_FPGA” is significantly lower than that of “GPU_CPU” plus “CPU_FPGA”, proving the efficiency of control plane offloading proposed in FpgaNIC.

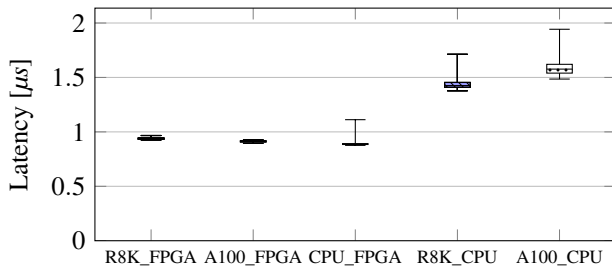


Figure 3: Control plane latency comparison. X_Y refers to the device “X” accesses the device “Y”. R8K refers to RTX 8000 GPU, and A100 refers to A100 GPU. Whiskers show the 1st and 99th percentile.

Effect of Data Plane Offloading. We examine the effect of data plane offloading by measuring the throughput when the FPGA issues a DMA read/write operation to GPUs. Each operation transfers 4GB of data between the FPGA and the GPU memory. Figure 4 illustrates the achievable throughput with varying burst size, which is associated with the length of a DMA operation. It is interesting to observe that DMA read and write operations reach peak throughput at different burst sizes: 512B for read and 8K for write, indicating that we need to carefully choose the right DMA size to saturate the PCIe P2P bandwidth between FPGA and GPU. As with latency, a newer GPU class leads to much higher PCIe throughput. For example, a DMA read operation to an A100 GPU yields 12.6GB/s, close to the maximum possible PCIe bandwidth.

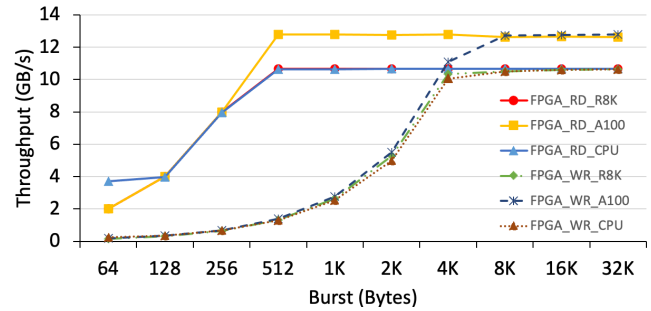


Figure 4: PCIe P2P throughput between FPGA and GPU

3.2.2 Hardware Network Transport

Next, we measure the throughput and latency of the hardware network transport (§2.4).

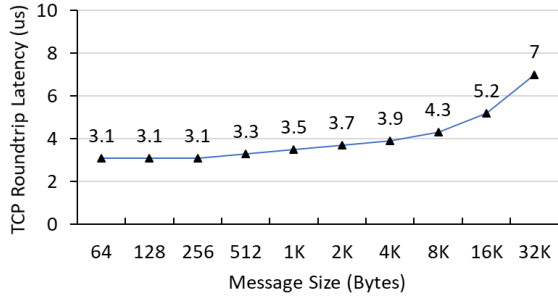
Latency. We measure the the round-trip time (RTT) between two FPGAs connected via the network switch. Figure 5a shows the RTT with varying message size. The most striking result is that the TCP latency is in microseconds, instead of milliseconds, demonstrating the advantages of offloading to a SmartNIC instead of using the CPU for communication. For messages smaller than 1 KB, the RTT latency (roughly 3.1us) is dominated by the physical communication path (the Ethernet switch introduces an additional hop with roughly 1us latency).

Throughput. We measure as well the throughput between two network transports with varying packet size and varying number of connections. Figure 5b shows the observed throughput by sending out a total of 1GB from one transport to the other with varying packet size and number of connections. We observe that the number of connections does not affect the achievable throughput under the same packet size, indicating that FpgaNIC is able to efficiently support multi-connection communication. For small packets, the throughput is low due to the fixed overhead, i.e., the 40B header, per packet and the turnaround cycles to process each packet. However, for larger packets, the achievable throughput is close to the 100Gbps channel capacity, demonstrating that FpgaNIC efficiently uses the available network bandwidth.

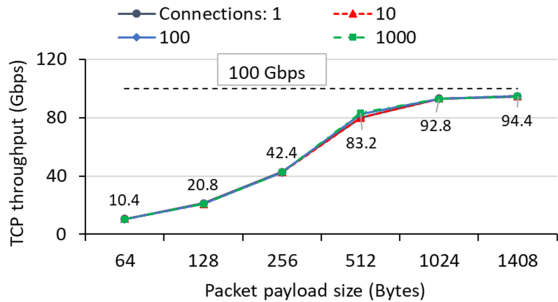
3.3 Evaluation of the Direct Model

We evaluate the throughput of FpgaNIC used in direct mode. The experiment involves sending data from one GPU to a remote GPU through the corresponding FPGAs using the direct model path: GPU-PCIe-FPGA-network-FPGA-PCIe-GPU.

Effect of Slot Size. We examine the effect of the slot size (W) of the circular buffer for each connection (§A.1.2). The slot size determines the size of the DMA operation between an FPGA and a GPU. Figure 6a illustrates the throughput with varying slot size. We have two observations. First, a sufficiently large slot size leads to saturated throughput. A



(a) Round-trip latency with varying message size



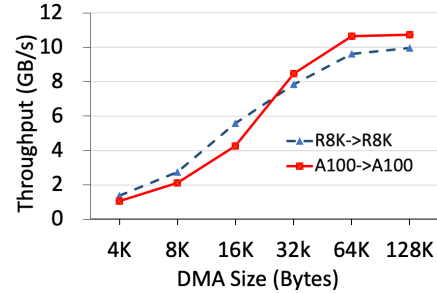
(b) Throughput with varying packet size and connections

Figure 5: 100Gb TCP stack: latency and throughput

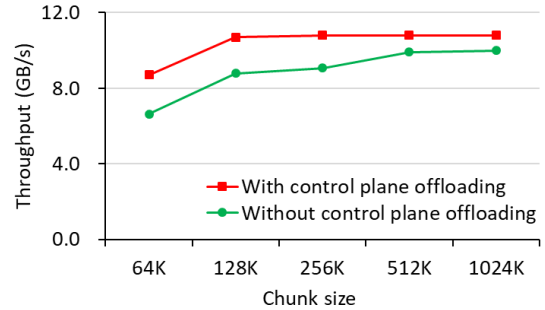
small slot size (<64KB) leads to lower throughput since it leads to low DMA engine utilization (Figure 4). Second, the network bandwidth between A100 GPUs is higher than that between RTX 8000 GPUs, as the slow PCIe speed between a RTX 8000 GPU and the FPGA becomes the bottleneck of network bandwidth (Figure 4).

Effect of Control Plane Offloading on Slot Size. We examine the effect of control plane offloading on different slot sizes. Without control plane offloading, we need to use CPU to trigger the DMA operation after executing a CUDA kernel that copies a chunk in the “GPU user” layer into the *send buffer* in the “GPU kernel” layer, leading to one kernel invocation per chunk. Intuitively, such frequent kernel invocations lead to significant overhead when the chunk size or the transfer size is not large. Figure 6b illustrates the throughput comparison with and without control plane offloading under different chunk size, when the data transfer size is 1GB. We observe that control plane offloading can lead to obviously higher throughput than the implementation without control plane offloading. Moreover, a smaller chunk size leads to higher throughput improvement, because control plane offloading eliminates more CUDA kernel invocations.

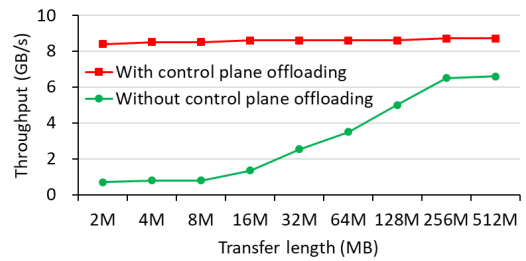
Effect of Control Plane Offloading on Transfer Size. We examine the effect of control plane offloading on different transfer sizes. Figure 6c illustrates the throughput comparison with and without control plane offloading under different transfer size, when the chunk size is 64KB. We observe that when the transfer length is smaller, control plane offloading leads to significant throughput improvement over the case without control plane offloading, whose performance is domi-



(a) Effect of DMA size with control plane offloading



(b) Effect of control plane offloading on chunk size



(c) Effect of control plane offloading on transfer size

Figure 6: Throughput of GPU-centric networking

nated by the kernel invocation overhead and context switch. In contrast, control plane offloading can remove these overheads by triggering doorbell registers from within a CUDA kernel, rather other from the host CPU.

3.4 Evaluation of the Off-path Model

In this subsection, we evaluate the performance of FpgaNIC-enhanced AllReduce on a distributed pool of eight GPUs, as shown in Figure 2. When accelerating AllReduce, we configure FpgaNIC in an off-path model and offload the AllReduce engine to the FPGA. In the following, we present the baseline and the corresponding performance comparison.

Baseline. The experimental platform used as a baseline is similar to Figure 2, except that FpgaNIC in each server is replaced with a Mellanox ConnectX-5 100Gbps MT27800 NIC with RoCE and GPUDirect enabled. We use NVIDIA Collective Communication Library (NCCL) [50] which provides state-of-the-art collective communication primitives, e.g., AllReduce, over distributed Nvidia GPUs.

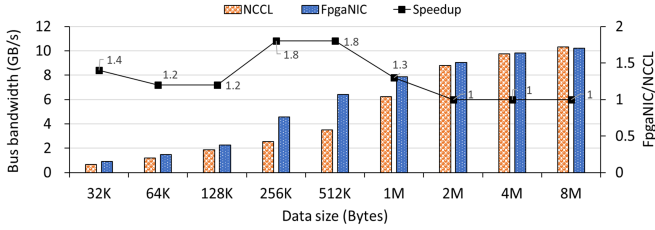


Figure 7: Effect of data size under eight nodes

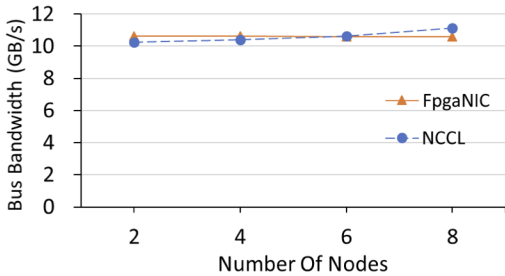


Figure 8: Effect of node number with 64MB data size

Comparison Metric. To demonstrate the performance of AllReduce, we introduce the metric *bus bandwidth* [51], which is calculated to be *algorithm bandwidth* times $2 * (N - 1) / N$, where algorithm bandwidth is calculated to be the data size divided by the elapsed time and N is the number of nodes. The elapsed time is estimated to be the average time of all the involved nodes in five rounds [9].

Effect of Data Size. We examine the effect of data size when performing AllReduce. Intuitively, a large data size easily leads to a saturated throughput because we hit the bandwidth limits of the underlying channels. Figure 7 illustrates the bus bandwidth comparison between FpgaNIC and NCCL in a cluster with 8 nodes. FpgaNIC leads to up to 2.5x speedup over NCCL, because the AllReduce engine in FpgaNIC efficiently overlaps the operations of the PCIe DMA, network transport, and FPGA memory. Moreover, FpgaNIC does not consume any GPU/CPU cycles, freeing up these precious computing resources for other important tasks. FpgaNIC reaches the theoretical bus bandwidth when the data size is larger than 8MB, indicating that FpgaNIC’s AllReduce implementation is using all resources efficiently. Finally, when data size is small (<1MB), the speedup is up to 2.5x, due to the faster transition between states in FpgaNIC (Table 8) when compared to the same operation being implemented on GPUs.

Impact of Systems Size. We examine the effect of number of nodes on the AllReduce performance under 64MB data size. Figure 8 shows how both FpgaNIC and NCCL reach the theoretical bus bandwidth with an increasing number of nodes. However, NCCL needs a quite amount of CPU/GPU computing cycles to realize, while FpgaNIC does not.

Discussion. In the context of distributed AI model training, these results indicate that only offloading the AllReduce engine will not be able to fully harvest FpgaNIC’s potential. We can offload not only the communication functions (i.e., the

AllReduce engine) but also part of the learning engine such as the compressor (e.g., compression engine) and optimizer (e.g., Adam engine) to FpgaNIC, such that the entire communication part of training is offloaded to minimize the communication overhead for GPUs. Since these engines can easily achieve line-rate throughput, plenty of interesting trade-offs in the design of distributed learning, e.g., sync vs. async, can be revisited by using FpgaNIC. We leave this idea to future work.

3.5 Evaluation of the On-path Model

We finally evaluate the performance of FpgaNIC-enhanced HLL, when FpgaNIC is configured in an on-path model. The cardinality is calculated when the data stream has been transferred. The goal of this experiment is to verify whether the HLL module within an FPGA can act as a bump in the wire.

The baseline, labelled “write”, is to feed data to a GPU without processing the data in the FPGA. The GPU receives the data from the FPGA and stores it in the current *block* in GPU memory, while at the same time performing HLL on the previous block, overlapping data transfer with cardinality calculation at the block granularity. Table 6 illustrates that at least 8 SMs, in terms of 8 thread blocks and 512 threads per a thread block, are required to consume 100Gbps data stream (packet payload size: 1408 bytes) on an A100 GPU, when the block size is no smaller than 256K. Moreover, when the block size is smaller than 128K, an A100 GPU is not able to consume the data stream, as such a small block size cannot fully utilize GPU’s processing parallelism.

Table 6: Number of required GPU SMs w.r.t block size

Block size	<=128K	256K	512K	1024K
Number of A100 SMs	>256	8	8	8

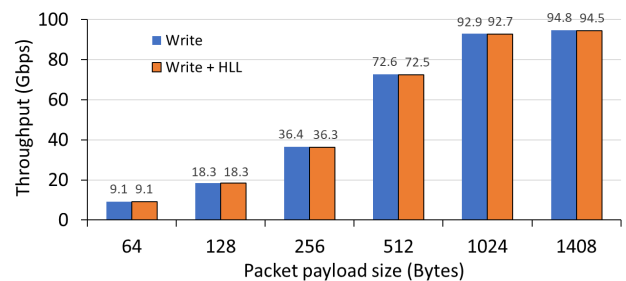


Figure 9: Performance of HLL with and without offloading. HLL with offloading does not affect the overall throughput, but saves at least 8 A100 GPU SMs, required by HLL without offloading, to consume 100 Gbps HLL data stream.

Figure 9 illustrates that FpgaNIC-enhanced HLL is able to achieve similar throughput as the baseline under various packet payload size, where the block size is 256K. It indicates that offloading HLL does not block the incoming data stream and introduces negligible latency. More important, FpgaNIC-enhanced HLL does not require any GPU compute power,

Table 7: Comparison of FpgaNIC with existing SmartNICs from industry. ✓ indicates full support, ✗ indicates no support.

	Programmable flow processing	Targeted applications	CPU-centric	GPU-centric
Broadcom [7]	✓	Virtualization, storage, NFV	✓	✗
Pensando [53]	✓	Storage, security	✓	✗
Netronome [49]	✓	SDN-controlled server-based networking	✓	✗
Intel IPU [23,24]	✓	Cloud, storage, security	✓	✗
FpgaNIC	✓	AI model training	✗	✓

e.g., at least 8 A100 SMs, which can be used in other computing task. Moreover, Table 2 shows that FpgaNIC-enhanced HLL takes a small amount of FPGA resources. Therefore, in the context of FPGA+GPU co-processing, it is clearly more efficient to offload HLL onto FpgaNIC, rather than processing HLL on the GPU.

4 Related Work

To our knowledge, FpgaNIC is the first FPGA-based GPU-centric 100Gbps SmartNIC that addresses the bottleneck limitations introduced by the use of conventional CPUs or small cores (ARM) in SmartNICs.

FPGA-augmented SmartNICs. Several commercial systems [10, 21, 22, 45, 55, 79] feature an FPGA within a SmartNIC. The closest work is from Mellanox Innova [45] that features an FPGA in its SmartNIC to accelerate offloaded compute-intensive applications, while PCIe and network interfaces are handled by a NIC ASIC ConnectX-5. The FPGA is connected with the NIC ASIC via a PCIe interface and therefore acts as an additional PCIe endpoint. The FPGA is entirely dedicated to the user’s application logic. In contrast, FpgaNIC implements all functionalities, including networking and PCIe, within a powerful FPGA, enabling a large design space exploration of SmartNIC architecture, while Innova provides limited architectural flexibility due to how the FPGA is connected.

GPU-FPGA Communication. Previous work [6, 64] has implemented GPUDirect RDMA on an FPGA to directly access GPU memory, but not allowing the GPU to trigger doorbell registers within an FPGA. In contrast, FpgaNIC allows GPUDirect RDMA and the GPU to trigger registers within an FPGA, and is an FPGA-based SmartNIC that allows large design space exploration of SmartNIC architecture.

Acceleration using FPGA-based SmartNICs. Most previous work [2, 3, 8, 10, 13, 14, 14, 17, 26, 35, 36, 36, 37, 59, 62, 65] features an FPGA on SmartNICs to offload data processing to the network from the host CPU. In contrast, FpgaNIC is an FPGA-based full-stack SmartNIC that mainly targets compute task offloading from local GPUs which require a more complex system than offloading for CPUs. For example, we can offload partial G-TADOC [76, 77] that is a novel optimization to perform compressed data direct processing onto FpgaNIC to maximize the performance of distributed system under efficient FPGA-GPU co-processing.

Multicore-based SmartNICs. There is also a lot of work

done [12, 16, 29, 41, 42, 44, 47, 48, 54, 56, 63, 66, 70] on SmartNIC built upon a wimpy RISC cores plus hardware engines to accelerate dedicated functionality such as compression. These RISC cores are used to both process packets as well as to implement “smart” functions instead of using the host CPUs. Such an approach inevitably suffers from load interference since packet processing and the smart functions have to compete for the shared resources, e.g., the last level cache and memory bandwidth. In contrast, FpgaNIC implements an GPU-centric SmartNIC on an FPGA.

On-going SmartNICs in Industry. Besides NVIDIA’s DPU, we compare FpgaNIC with other SmartNICs from industry, as shown in Table 7. Broadcom offers the Stingray SmartNIC [7], which features a ARM 8-core CPU for control-plane management and P4-like TruFlow packet processing engine for data-plane processing, targeting various applications such as virtualization, storage, and NFV. Pensando has a DPU architecture [53] that features an ARM CPU and a P4 processor for data-plane packet processing, targeting various applications such as security and storage. Netronome provides the NFP4000 Flow Processor architecture [49] that features a ARM CPU, 48 packet processing cores, and 60 P4-programmable flow processing cores for data-plane processing, targeting the SDN-controlled server-based networking application. Intel presents the FPGA-based IPU (Infrastructure Processing Unit) that consists of a MAX 10 FPGA for control-plane management and an Arria 10 FPGA for data-plane processing [23], and uses an ASIC IPU whose architecture is not publicly documented [24]. Fungible has a DPU [15] featuring multiple PCIe endpoints, TrueFabric for networking, and specialized engines such as compression and EC/RAID to address inefficient data-centric computation within a node and inefficient interchange between nodes, targeting various applications such as virtualization, cloud storage, and data analytics. All these systems are CPU-centric in that they are designed to complement the CPU. In several cases, they suffer from the bottleneck problem pointed out above that prevents them from being able to operate at line rate. In contrast, FpgaNIC is an FPGA-based GPU-centric SmartNIC that targets various applications such as AI model training and security and specifically designed to operate at line rate which also means that it might not be suitable for operations that would significantly impair the flow of network packets (such as blocking operations or computations generating large amounts of intermediate state).

5 Insights and Implications of FpgaNIC

In this section, we discuss three interesting properties regarding FpgaNIC.

High Performance of On-NIC Computing Module. An increasing amount of SmartNIC solutions intend to remove the conventional CPU from the data-path (e.g., Microsoft Catapult). However, they either do not exploit the possibilities of direct communication between the FPGA and the GPU, or use small CPUs (ARM cores) that cannot process at line rate to impose additional hops within the NIC to implement the smart functionality (e.g., Bluefield-2). Furthermore, commercially available multi-core SmartNICs, such as BlueField-2, fail to process 100Gbps network traffic at line rate with its embedded CPU, which is capable of doing control-plane management only. The embedded CPU in Bluefield-2 is overwhelmed by trying to stage a 100Gbps data stream coming from the network. In contrast, FpgaNIC provides a 100Gbps data-path accelerator for distributed computing over GPUs, and thus enables a large design space exploration around SmartNIC for GPU-based applications. The key aspect of FpgaNIC is that it can process data at line rate as it comes from the network, something that other systems cannot do, because this requires to insert the accelerator in the data path, which cannot be done with conventional hardware (running conventional software) but can be done with FPGAs. To do so, FpgaNIC only consumes roughly 20% of the FPGA resources (marked in blue) to implement the NIC architecture (100Gbps hardware network transport and GPU communication stack) in a half-length, half-height FPGA board (Alveo U50), as shown in Table 5. It implies that the majority of the FPGA resources can be dedicated to on-NIC computing for SmartNIC functionality. Moreover, U50 has High Bandwidth Memory (HBM) which can be used to implement functionality with more intermediate states as memory access does not become the bottleneck. Therefore, FpgaNIC allows the offloading of compute-bound and memory-intensive tasks from multiple tenants (e.g., like in [39]) onto a mid-size FPGA.

Performance Guarantee and Isolation. Many multicore-based SmartNICs use small CPU cores for in-network computing. On these CPUs is really hard to provide performance guarantee and isolation due to insufficient CPU processing abilities and interference across tasks. We have shown that FpgaNIC is able to guarantee performance and isolation from two perspectives. From a compute's perspective, FpgaNIC provides dedicated hardware resources for each offloaded compute task, leading to a strict performance guarantee and perfect performance isolation. From a memory's perspective, U50 features 2-channel HBMs [71, 75] with 32 independent memory channels, each of which provides up to 13.6GB/s of memory throughput [20, 68]. This guarantees that each offloaded compute task is able to gain exclusive control over the assigned memory channels, without interfering with other offloaded compute tasks and the NIC infrastructure, which op-

erates on dedicated hardware resources to guarantee line-rate network throughput.

Medium Programmability. Programming FPGAs using a Hardware Description Language (HDL), is error-prone and difficult to debug, limiting the adoption of FPGAs by system programmers. When using FpgaNIC, we intentionally ensure that it can be programmed using C++-based HLS (High Level Synthesis), to make it easier to use for software programmers, where HLS is the highest level of abstraction commercially available for programming FPGAs. To let FpgaNIC support both HDL and HLS, FpgaNIC's interface mainly leverages the stream type in HLS, i.e., AXI stream in HDL, for better compatibility. In future work, we intend to raise the level of abstraction further by developing a comprehensive framework such that users without hardware design experience can easily leverage FpgaNIC to accelerate distributed GPU-powered applications by automatically identifying offloaded functionalities via an FPGA-aware performance analysis framework [69] for maximum performance and high programmability.

6 Conclusion

Inspired by the fact that there is no SmartNIC designed for GPUs, we present FpgaNIC, a full-stack FPGA-based GPU-centric 100Gbps SmartNIC that allows a large design space exploration around SmartNICs for accelerating applications running on distributed GPUs. FpgaNIC enables direct data communication to local GPUs via PCIe P2P communication, enables local GPUs to directly manipulate the FPGA, provides reliable network communication with remote nodes, and enables on-NIC computing module to process the data from network at line rate. FpgaNIC can be efficiently used in three SmartNIC modes: direct, off-path, and on-path, to accelerate a broad range of GPU-powered distributed applications, such as Deep Learning model training. FpgaNIC is open-source to encourage further development and research in GPU-centric applications (Github: <https://github.com/RC4ML/FpgaNIC>).

Acknowledgement. We thank our shepherd and anonymous reviewers for their constructive suggestions. We are grateful to the AMD-Xilinx University Program for the donation of some of the AMD-Xilinx FPGAs used in the experiments. The work is supported by the following grants: the National Key R&D Program of China (Grant No. 2020AAA0103800), the Fundamental Research Funds for the Central Universities 226-2022-00151 and 226-2022-00051, Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study (SN-ZJU-SIAS-0010).

References

- [1] Alexey Kopytov. sysbench. <https://github.com/akopytov/sysbench>, 2020.

- [2] Catalina Alvarez, Zhenhao He, Gustavo Alonso, and Ankit Singla. Specializing the Network for Scatter-Gather Workloads. In *SOCC*, 2020.
- [3] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling Programmable Transport Protocols in High-Speed NICs. In *NSDI*, 2020.
- [4] Baidu. baidu-allreduce. <https://github.com/baidu-research/baidu-allreduce>, 2016.
- [5] M. Banikazemi, V. Moorthy, and D. K. Panda. Efficient collective communication on heterogeneous networks of workstations. In *ICPP*, 1998.
- [6] R. Bittner and E. Ruf. Direct GPU/FPGA Communication via PCI Express. In *ICPP Workshops*, 2012.
- [7] Broadcom. Stingray PS250 2x50-Gb High-Performance Data Center SmartNIC. <https://docs.broadcom.com/doc/PS250-PB>, 2019.
- [8] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *OSDI*, 2020.
- [9] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. K. Panda. Omb-gpu: A micro-benchmark suite for evaluating mpi libraries on gpu clusters. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, 2012.
- [10] A. M. Caulfield, E. S. Chung, A. Putnam, et al. A Cloud-scale Acceleration Architecture. In *MICRO*, 2016.
- [11] M. Cho, U. Finkler, M. Serrano, D. Kung, and H. Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *IBM Journal of Research and Development*, 2019.
- [12] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *NSDI*, 2014.
- [13] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *ATC*, 2019.
- [14] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, et al. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, 2018.
- [15] Fungible. FUNGIBLE F1 DATA PROCESSING UNIT. <https://www.fungible.com/wp-content/uploads/2020/08/PB0028.01>.
- [16] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. In *SIGCOMM*, 2020.
- [17] Zhenhao He, Dario Korolija, and Gustavo Alonso. EasyNet: 100 Gbps Network for HLS. In *FPL*, 2021.
- [18] Stefan Heule, Marc Nunkesser, and Alexander Hall. HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm. In *CEDT*, 2013.
- [19] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell. sPIN: High-performance Streaming Processing in the Network. In *SC*, 2017.
- [20] Hongjing Huang, Zeke Wang, Jie Zhang, Zhenhao He, Chao Wu, Jun Xiao, and Gustavo Alonso. Shuhai: A Tool for Benchmarking High Bandwidth Memory on FPGAs. *TC*, 2022.
- [21] Intel. Intel SmartNICs for Telecommunications. <https://www.intel.com/content/www/us/en/products/programmable/smart-nics-fpga-for-broadband-edge.html>, 2020.
- [22] Intel. Infrastructure Processing Units (IPUs) and SmartNICs. <https://www.intel.com/content/www/us/en/products/network-io/smartnic.html>, 2021.
- [23] Intel. Intel FPGA Programmable Acceleration Card N3000 for Networking. <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/po/intel-fpga-programmable-acceleration-card-n3000-for-networking.pdf>, 2021.
- [24] Intel. Intel Unveils Infrastructure Processing Unit. <https://www.intel.com/content/www/us/en/newsroom/news/infrastructure-processing-unit-data-center.html#gs.bdzkbc>, 2021.
- [25] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *NSDI*, 2016.
- [26] Wenqi Jiang, Zhenhao He, Shuai Zhang, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, and Gustavo Alonso. FleetRec: Large-Scale Recommendation Inference on Hybrid GPU-FPGA Clusters. In *KDD*, 2021.

- [27] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *OSDI*, 2020.
- [28] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. Raising the Bar for Using GPUs in Software Packet Processing. In *NSDI*, 2015.
- [29] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *ASPLOS*, 2016.
- [30] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *OSDI*, 2014.
- [31] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [32] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on FPGAs? In *OSDI*, 2020.
- [33] A. Kulkarni, M. Chiosa, T. B. Preußer, K. Kara, D. Sidler, and G. Alonso. HyperLogLog Sketch Acceleration on FPGA. In *FPL*, 2020.
- [34] N. T. Kung and R. Morris. Credit-based Flow Control for ATM Networks. *IEEE Network*, 1995.
- [35] N. Lazarev, N. Adit, S. Xiang, Z. Zhang, and C. Delimitrou. Dagger: Towards Efficient RPCs in Cloud Microservices With Near-Memory Reconfigurable NICs. *IEEE Computer Architecture Letters*, 2020.
- [36] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *SOSP*, 2017.
- [37] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *SIGCOMM*, 2016.
- [38] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. Schwing, H. Esmaeilzadeh, and N. S. Kim. A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks. In *MICRO*, 2018.
- [39] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *OSDI*, 2020.
- [40] Jianshen Liu, Carlos Maltzahn, Craig Ulmer, and Matthew Leon Curry. Performance Characteristics of the BlueField-2 SmartNIC. *arXiv preprint arXiv:2105.06619*, 2021.
- [41] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs Using IPipe. In *SIGCOMM*, 2019.
- [42] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-efficient microservices on smartnic-accelerated servers. In *ATC*, 2019.
- [43] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-path transport for RDMA in datacenters. In *NSDI*, 2018.
- [44] Mellanox. Mellanox BlueField SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf, 2019.
- [45] Mellanox. Mellanox innova-2flex. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf, 2020.
- [46] Michael Cooney. Speed Race: Just as 400Gb Ethernet Gear Rolls Out, an 800GbE SPEC is Revealed. <https://www.prnewswire.com/news-releases/400gbe-to-drive-the-majority-of-data-center-ethernet-switch-bandwidth-within-five-years-forecasts-crehan-research-300587873.html>, 2020.
- [47] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. Acceltcp: Accelerating network applications with stateful TCP offloading. In *NSDI*, 2020.
- [48] Craig Mustard, Fabian Ruffy, Anny Gakhokidze, Ivan Beschastnikh, and Alexandra Fedorova. Jumpgate: In-Network Processing as a Service for Data Analytics. In *HotCloud*, 2019.
- [49] Netronome. NFP-4000 Theory of Operation. https://www.netronome.com/static/app/img/products/silicon-solutions/WP_NFP4000_T00.pdf, 2016.
- [50] Nvidia. NVIDIA NCCL. <https://developer.nvidia.com/nccl>, 2016.

- [51] Nvidia. Performance reported by NCCL tests. <https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md>, 2018.
- [52] NVIDIA. Developing a Linux Kernel Module using GPUDirect RDMA. <https://docs.nvidia.com/cuda/gpudirect-rdma/index.html>, 2020.
- [53] Pensando. Pensando DSC-25 Distributed Services Card. <https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-25-Product-Brief.pdf>, 2020.
- [54] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *OSDI*, 2018.
- [55] Andrew Putnam, Adrian M Caulfield, Eric S Chung, et al. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *ISCA*, 2014.
- [56] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. Clara: Performance Clarity for SmartNIC Offloading. In *HotNets*, 2020.
- [57] M. Ruiz, D. Sidler, G. Sutter, G. Alonso, and S. López-Buedo. Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack. In *FPL*, 2019.
- [58] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *SOSP*, 2019.
- [59] Ran Shu, Peng Cheng, Guo Chen, Zhiyuan Guo, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. Direct Universal Access: Making Data Center Resources Available to FPGA. In *NSDI*, 2019.
- [60] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley. Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware. In *FCCM*, 2015.
- [61] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *SIGMOD*, 2017.
- [62] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulकर्णी, and Gustavo Alonso. StRoM: Smart Remote Memory. *EuroSys*, 2020.
- [63] Brent Stephens, Aditya Akella, and Michael M. Swift. Your Programmable NIC Should Be a Programmable Switch. In *HotNets*, 2018.
- [64] Y. Thoma, A. Dassatti, and D. Molla. FPGA2: An Open Source Framework for FPGA-GPU PCIe Communication. In *ReConFig*, 2013.
- [65] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. The Case For In-Network Computing On Demand. In *EuroSys*, 2019.
- [66] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers. In *ASPLOS*, 2020.
- [67] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and Generic Collectives for Distributed ML. In *MLSys*, 2020.
- [68] Z. Wang, H. Huang, J. Zhang, and G. Alonso. Shuhai: Benchmarking High Bandwidth Memory On FPGAs. In *FCCM*, 2020.
- [69] Zeke Wang, Bingsheng He, Wei Zhang, and Shunning Jiang. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *HPCA*, 2016.
- [70] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *OSDI*, 2020.
- [71] Mike Wissolik, Darren Zacher, Anthony Torza, and Brandon Day. Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance, 2019.
- [72] Xilinx. XILINX ALVEO Adaptable Accelerator Cards for Data Center Workloads. <https://www.xilinx.com/products/boards-and-kits/alveo.html>, 2010.
- [73] Xilinx. Alveo U280 Data Center Accelerator Card Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds963-u280.pdf, 2019.
- [74] Xilinx. Alveo U50 Data Center Accelerator Card Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds965-u50.pdf, 2019.
- [75] Xilinx. AXI High Bandwidth Memory Controller v1.0, 2019.
- [76] Feng Zhang, Zaifeng Pan, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. G-TADOC: Enabling Efficient GPU-based Text analytics without Decompression. In *ICDE*, 2021.
- [77] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. TADOC: Text Analytics Directly on Compression. *VLDBJ*, 2021.

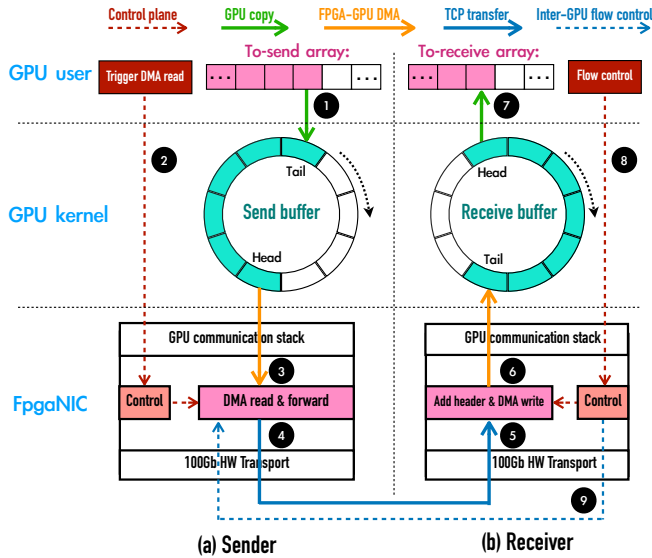


Figure 11: GCN between a sender and a receiver

GPU Kernel Layer. To manage incoming and outgoing traffic, we implement a *send buffer* and a *receive buffer*, conceptually two circular buffers, for each established connection. The key role of either buffer is to provide a staging option at the GPU memory exposed to other PCIe devices, as not all the memory space is visible to other PCIe devices on Quadro GPUs. Since the total exposed memory size is 220MB, we allocate 100MB to the send buffers and 100MB to the receive buffers, while the remaining 20MB is reserved for internal use. The number of supported connections is N , so each connection has a circular buffer of size $M = 100/N$ MB. We also split a circular buffer into F slots, each of which containing $W = \frac{M}{F}$ MB GPU memory space.¹¹

GPU User Layer. In the GPU user layer, calling a `send()` or a `receive()` function will launch a *data-mover* kernel that leverages Streaming Multiprocessing (SM)¹² to move data between the GPU user and kernel layers such that the speed of the data mover matches the DMA read/write speed.

A.1.3 Handshake Protocol of GCN

We demonstrate how the handshake protocol works in GCN. The key idea is to directly leverage the TCP stack on the FPGA (§2.4) via the control plane offloaded to GPUs. The handshake process consists of the following three steps.

First, each side creates a GPU-aware context by calling `create_socket_context`, which specifies the number N of supported TCP connections (e.g., 8), the GPU send/recv buffer size of each connection (e.g., 12.5MB), the initial address of control plane.

¹¹Nevertheless, FpgaNIC can be easily extended to support dynamic buffer size with slight modifications in the GPU kernel layer.

¹²In our experiment, a Streaming Multiprocessing (SM) provides more than enough throughput on both Quadro and Tesla GPUs. Each SM consists of 64 GPU cores. RTX8000 has 72 SMs while A100 has 108.

Second, each side creates a socket (`sockfd`) using the function `socket`, which launches a GPU kernel with only one thread that will apply for one free TCP connection slot in the FPGA 100Gb TCP stack.

Third, the server will listen to the socket `sockfd` by setting the listen-port register `listen_port` and then triggering the doorbell register `listen_start` (Table 4). Then, the server initiates the function `accept` to wait for an incoming connection from a client. The client calls the function `connect` with two parameters `conn_port` and `conn_ip` to specify the destination IP address and port. Once a connection is established, a client and a server can proceed to exchange data.

A.1.4 Send/Recv Functions of GCN

We now describe the implementation details of the two-sided communication between distributed GPUs by explaining the overall data and control flow shown in Figure 11.

Data Flow. The sender splits the “to-send array” into chunks, each of which has the size of W MB. For each chunk, we perform the following five steps. First, we employ a data-mover kernel that occupies a GPU SM, in terms of a thread block with 1024 threads, to copy a chunk in the “GPU user” layer into the “tail” slot in the *send buffer* in the “GPU kernel” layer (❶). Second, the sender kernel triggers a doorbell register within an FPGA to start a DMA read operation (❷). Third, the *DMA read* module reads the data stream from the “head” slot in the *send buffer* (❸), and then forwards to the 100Gb TCP stack (❹). Fourth, the receiver accepts the data stream from its 100Gb TCP stack (❺), and then adds a header and a trailer to the data stream and forwards it to the “tail” slot in the *receive buffer* in the “GPU kernel” layer (❻). Fifth, the receiving GPU kernel monitors the “head” slot and leverages a data-mover kernel that also occupies a GPU SM to copy the data in the “tail” slot to the destination chunk in the to-*receive array* in the “GPU user” layer (❼).

Flow Control. Reliable communication (goal D1) is achieved through a simple credit-based flow control [34] over each TCP connection, so as to avoid potential congestion at a slow receiving GPU receiving a heavy traffic load. At the beginning, the sender has a full credit of M MB to leverage. If we send data from the to-*send array* to the tail slot in the *send buffer* (❶), the corresponding credits are consumed, and the data in the *send buffer* will be safely delivered to the *receive buffer* on the other side. When the receiver copies the data from the head slot to the to-*receive array* in the “GPU user” layer, and then accumulates the amount M_r of correctly received data, where M_r is initialized to be 0. Once the ratio of M_r to M is over a threshold (Th), the receiver sends back a credit with $Th \times M$ bytes to the sender, indicating $Th \times M$ bytes of data have been correctly received. To do so, the receiver triggers a doorbell register (`consumed_bytes`) specified in the “control” module (❸), and then forms a flow-control packet to the sender (❾). After the sender receives the credits, the sender can proceed to send $Th \times M$ additional bytes.

Table 8: State transition of AllReduce within FpgaNIC. “ x/y ” means that x is input and y is output, where G refers to the communication with a GPU, E refers to the communication with the 100Gb TCP stack. “ $(G + E)$ ” means that we perform the reduction on the data from GPUs (G) and the data from the 100Gb TCP stack (E), and store the reduced result in on-board memory. “ (E, G) ” means that the data is read from on-board memory and forwarded to the next GPU via 100Gb TCP stack (E) and GPUs (G). E_i^j indicates the *subarray*[i] has already been accumulated j times, where $1 \leq j \leq 4$. When j is 4, E^4 and G^4 are the final reduced result sent to the next GPU via 100Gb TCP stack and to local GPU, respectively.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
FPGA 0	$G_0^1/-$	$(G_3^1 + E_3^1)/E_0^1$	$(G_2^1 + E_2^2)/E_3^2$	$(G_1^1 + E_1^3)/E_2^3$	$E_0^4/(E_1^4, G_1^4)$	$E_3^4/(E_0^4, G_0^4)$	$E_2^4/(E_3^4, G_3^4)$	$-/G_2^4$
FPGA 1	$G_1^1/-$	$(G_0^1 + E_0^1)/E_1^1$	$(G_3^1 + E_3^2)/E_0^2$	$(G_2^1 + E_2^3)/E_3^3$	$E_1^4/(E_2^4, G_2^4)$	$E_0^4/(E_1^4, G_1^4)$	$E_3^4/(E_0^4, G_0^4)$	$-/G_3^4$
FPGA 2	$G_2^1/-$	$(G_1^1 + E_1^1)/E_2^1$	$(G_0^1 + E_0^2)/E_1^2$	$(G_3^1 + E_3^3)/E_0^3$	$E_2^4/(E_3^4, G_3^4)$	$E_1^4/(E_2^4, G_2^4)$	$E_0^4/(E_1^4, G_1^4)$	$-/G_0^4$
FPGA 3	$G_3^1/-$	$(G_2^1 + E_2^1)/E_3^1$	$(G_1^1 + E_1^2)/E_2^2$	$(G_0^1 + E_0^3)/E_1^3$	$E_3^4/(E_0^4, G_0^4)$	$E_2^4/(E_3^4, G_3^4)$	$E_1^4/(E_2^4, G_2^4)$	$-/G_1^4$

Discussion. The design matches the goals we established the beginning, which dictate many of the architectural decisions. To ensure reliable communication (goal **D1**), we implement a credit-based flow control (§A.1.4) on the GPU side. To simplify programming (goal **D2**), the GPU-aware socket-like functions are executed sequentially by leveraging the default CUDA stream, which is transparent to programmers. Nevertheless, our APIs also allow programmers to explicitly specify CUDA streams to maximize execution overlap between kernels. To minimize overhead (goal **D3**), GCN uses at most 220MB of the GPU memory for the communication, and a GPU SM only when a send() or a receive() function is active. Moreover, each handshake function launches a GPU kernel with only one active thread. Finally, to support a wide range of GPU classes (goal **D4**), GCN only exposes 220MB GPU memory, which is allowed in all supported Nvidia GPUs.

A.2 Off-path SmartNIC: AllReduce

To illustrate the the off-path model of FpgaNIC, we implement a use case from HPC and AI applications: a collective communication primitive AllReduce [4, 11, 50] operating on the data residing in a distributed pool of GPUs. In particular, we implement a ring-based AllReduce algorithm [4, 50] as it provides high performance while having a simple communication flow that fits well within an FPGA. The communication pattern is as follows. Assume there are P GPUs and each GPU divides its own array for AllReduce into P subarrays. The p -th GPU receives *subarray*[i] from the $(p - 1)$ -th GPU, performs a reduction operation on the received *subarray*[i] and its local *subarray*[i], and then sends the reduced result to the $(p + 1)$ -th GPU, where $0 \leq i, p < P$.

A.2.1 Overall Architecture of AllReduce

The AllReduce [5, 67] engine implements the entire logic in the ONC component, which is configured in an off-path model, as show in Figure 12. The engine concurrently operates on three components on the FPGA: the PCIe DMA operation (§2.3), the network stack (§2.4), and the on-board memory. The overall execution under FpgaNIC allows to overlap the access to these components to maximize throughput.

PCIe DMA Operation. The PCIe DMA operation is used transfer data between the FPGA and its local GPU by issuing a DMA operation within an FPGA directly to the GPU and without CPU intervention.

Network Stack. The network stack is used to communicate with remote GPUs through their corresponding FPGAs. In our ring implementation, data arrives the previous GPU and it is sent to the next GPU in the ring.

On-board Memory. The FPGA on-board memory is used to store intermediate results. The current reduced result is accumulated in on-board memory before being forwarded to the next GPU. While it is being forwarded, the next result is being calculated. Thus, the memory needs to provide sufficient bandwidth for simultaneously writing partial results and reading the previous result as it is being sent.

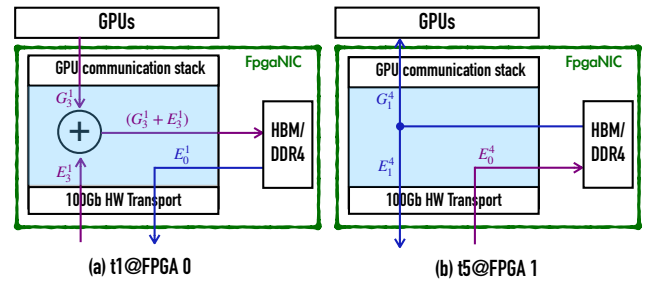


Figure 12: Architecture of AllReduce on the off-path model

A.2.2 Execution Flow of AllReduce on FpgaNIC

Table 8 illustrates the detailed execution flow of FpgaNIC-enhanced AllReduce with a concrete example over 4 distributed nodes, labelled FPGA i , where i is from 1 to 4. The execution is divided into eight steps ($t_0 \sim t_7$).

At step t_0 , FPGA i issues a DMA read operation to transfer its local *subarray*[i] in GPU memory to the FPGA’s memory (labelled G_i^1). At step t_1 , FPGA i issues a DMA read operation to read from its local *subarray*[$i-1$] (G_{i-1}^1) in GPU memory, receives E_{i-1}^1 from the previous GPU in the ring (i.e., arriving via the network), and then accumulates these two on-the-fly and finally stores the accumulated result in the FPGA memory (labelled $(G_{i-1}^1 + E_{i-1}^1)$). At the same time, FPGA i

forwards its local *subarray*[*i*] in FPGA memory to the next GPU, labelled E_i^1 . Figure 12(a) illustrates the data flow of FPGA 0. At step t_2 , FPGA *i* performs the reduction operation on its local G_{i-2}^1 in GPU memory and E_{i-2}^2 from the previous GPU, and then stores the accumulated result in FPGA memory (labelled $(G_{i-2}^1 + E_{i-2}^2)$). At the same time, FPGA *i* forwards its local E_{i-1}^2 from the FPGA memory to the next GPU. At step t_3 , FPGA *i* performs the reduction operation on G_{i-3}^1 from its local GPU memory and E_{i-3}^3 from the previous GPU, and then stores the accumulated result in FPGA memory, labelled $(G_{i-3}^1 + E_{i-3}^3)$. At the same time, FPGA *i* sends its local E_{i-2}^3 from FPGA memory to the next GPU in the ring. At step t_4 , FPGA *i* receives E_i^4 from the previous GPU and copies it to FPGA memory. At the same time, FPGA *i* sends $(G_{i-3}^1 + E_{i-3}^3)$ from the FPGA memory to the next GPU E_{i-3}^4 and writes it to its local GPU memory G_{i-3}^4 . At step t_5 , FPGA *i* receives E_{i-3}^4 from the previous GPU and copies into the FPGA memory. At the same time, FPGA *i* sends E_i^4 in the FPGA memory to both the next GPU E_i^4 and writes it to its local GPU memory G_i^4 . Figure 12(b) illustrates the data flow of FPGA 1. At step t_6 , FPGA *i* receives E_{i-2}^4 from the previous GPU and copies it to on-board memory. At the same time, FPGA *i* sends E_{i-3}^4 to the next GPU E_{i-3}^4 and writes it to its GPU memory G_{i-3}^4 . At the final step t_7 , FPGA *i* writes E_{i-2}^4 from the FPGA memory into its GPU memory G_{i-2}^4 which now has the final aggregated result.

Comparison with AllReduce on Innova. To illustrate the advantages of FpgaNIC’s design over existing commercial solutions, consider how the same AllReduce operation would work on Mellanox Innova. In Innova, the PCIe link connecting the FPGA to the rest of the system limits the overall throughput because the AllReduce engine on the FPGA is forced to interact with both the local GPU and the network through its PCIe X8 Gen4 endpoint. In such a design, the FPGA can consume data either from the GPU or from the network but not from both at the same time. We estimate that the overall throughput would be halved. Both Innova and FpgaNIC approaches do not involve any GPU cores during execution, freeing up GPU cores for other computing tasks.

A.3 On-path SmartNIC: HyperLogLog

To illustrate the on-path model of FpgaNIC, we use HyperLogLog (HLL) [18, 33] as an example application. HLL is widely used in data analytic applications to estimate the cardinality of data streams or of large data sets. In our case, HLL will work on the data as it flows from the network transport towards the GPU. The basic scenario is transferring data to be processed to the GPU and using the FPGA to compute the cardinality on the fly without adding any overhead.

The on-path model is similar to the direct model (labelled “direct”), except that the incoming data stream is forwarded to both the GPU and to the on-path module (HLL in this case) via the “op_in” port. We use an open source implementa-

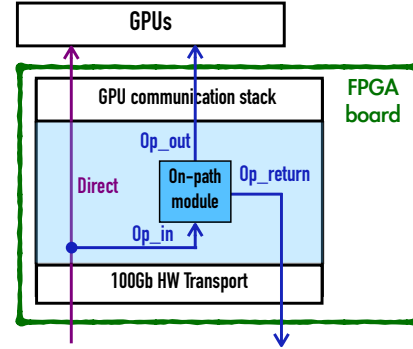


Figure 13: On-path model of FpgaNIC

tion of HLL [33] that is embedded into the ONC component (Figure 13). After the data stream has been consumed, the cardinality of the data set can be forwarded to the local GPU via the “op_out” port. The on-path model also allows the result to be sent back to the network through the “op_return” port. In such a case, FpgaNIC can be used as an independent, network attached accelerator that uses the on-path module to process data on behalf of a remote client.

7 Artifact

7.1 Abstract

This artifact provides the source code of FpgaNIC and scripts to reproduce the main experimental results. The experiments are run on a cluster consisting of eight 4U AMAX servers, connected with a Mellanox 100Gbps Ethernet SN2700 switch. Each server is equipped with two Intel Xeon Silver 4214 CPUs@2.20GHz, 128GB memory, FpgaNIC (i.e., a Xilinx Ultra-Scale+ FPGA), and a Nvidia RTX 8000 GPU, where the FPGA and the GPU have direct PCIe P2P communication. Two servers have an additional two A100 GPUs. FpgaNIC is implemented on Xilinx Alveo cards U50 or U280 with Vivado 2020.1.

7.2 Check-list

1. At least two nodes, each has a GPU that supports NVIDIA GPUDirect and the Xilinx U280 or U50 card.
2. Each FPGA card is connected to a 100Gbps Ethernet switch.
3. FPGA card and GPU are connected to the same PCIe switch.
4. Host OS: Linux 4.15.0-20-generic
5. Nvidia Driver Version: 450.51.05
6. CUDA Version: 11.0

Hugepages Setting. Make sure that each server has enabled Hugepages. If not, run the following commands.

1. \$ sudo apt install libboost-program-options-dev cmake
2. \$ sudo groupadd hugetlbf
3. \$ sudo getent group hugetlbf
4. \$ sudo adduser xxx hugetlbf
xxx is the user name you are using
5. Edit “/etc/sysctl.conf” and specify the number of pages you want to reserve.
6. \$ mkdir /media/huge
7. Add this line “hugetlbf /media/huge hugetlbf mode=1770,gid=1001 0 0” to “/etc/fstab”.
8. \$ reboot

7.3 Thee Steps to Run Experiments

There are three steps to run each experiment. Before running FpgaNIC, please clone the source code:

```
$ git clone https://github.com/RC4ML/FpgaNIC
```

7.3.1 Hardware: FPGA Bitstream

1. \$ mkdir build && cd build
2. \$ cmake ..
3. Make HLS IP core
\$ make installip
4. Create vivado project, add the hardware project option after *make*, as shown in Table 9.
\$ make pcie_benchmark
5. Now the hardware project is produced, generate bitstream using vivado and flush it to every FPGA card.
6. Every time you download the bitstream to the FPGA, you have to reboot the machine, do not forget to reinstall xdma driver and GDR driver (See Subsection 7.3.2).

Table 9: The options of hardware project

Project	Description
direct	To create direct model project
pcie_benchmark	To create PCIe benchmark project
tcp_latency	To create TCP latency benchmark project
tcp_benchmark	To create TCP throughput benchmark project
allreduce	To create off-path model project
hyperloglog	To create on-path model project

7.3.2 Software: Driver Installation

1. \$ cd FpgaNIC/driver
2. \$ make && sudo insmod xdma_driver.ko
3. \$ cd FpgaNIC/gdrcopy
4. \$ sudo ./insmod.sh
5. Note that you need to reinstall xdma driver and gdr driver every time you reboot your machine.

7.3.3 Software: Running Application Code

1. \$ cd FpgaNIC/sw && mkdir build && cd build
2. \$ cmake ../src
3. \$ make
4. \$ sudo ./dma-example -b 0
5. \$ Above command would report GPU read CPU memory latency, for more details, please refer to sw/README.md

Faster Software Packet Processing on FPGA NICs with eBPF Program Warping

Marco Bonola^{1,2}, Giacomo Belocchi^{1,3}, Angelo Tulumello^{1,3}, Marco Spaziani Brunella^{1,3}, Giuseppe Siracusano⁴, Giuseppe Bianchi³ and Roberto Bifulco⁴

¹Axbyrd, ²CNIT, ³University of Rome Tor Vergata, ⁴NEC Laboratories Europe

Abstract

FPGA NICs can improve packet processing performance, however, programming them is difficult, and existing solutions to enable software packet processing on FPGA either provide limited packet processing speed, or require changes to programs and to their development/deployment life cycle.

We address the issue with *program warping*, a new technique that improves throughput replacing several instructions of a packet processing program with an equivalent runtime programmable hardware implementation. Program warping performs static analysis of a packet processing program, described with Linux's eBPF, to identify subsets of the program that can be implemented by an optimized FPGA pipeline, the *warp engine*. Packets handled by the warp engine are eventually delivered to a regular eBPF program executor, along with their program context (registers, stack), to complete execution of those program parts that cannot be efficiently pipelined.

We prototype program warping on a 100Gbps FPGA NIC, extending hXDP, a state-of-the-art eBPF processor for FPGA, and measure its performance running 6 unmodified real-world eBPF programs, including deployed applications such as Katran and Suricata. Our prototype runs at 250MHz, uses less than 15% of the FPGA resources, and improves hXDP throughput by 1.2-3x in most cases, and up to 18x.

1 Introduction

Datacenter and telecom operators deploy FPGA NICs to handle network port speeds of 100Gbps or more, and to support heterogeneous applications and workloads [12, 21, 30]. In fact, these devices can host multiple accelerators [23], e.g., for radio signal processing, therefore providing a common hardware platform to address different scenarios [11].

Nonetheless, for network packet processing functions, such as firewalls or load balancers, FPGA NICs raise several challenges. First, programming FPGAs is difficult, often requiring dedicated teams of hardware specialists [15]. Second, it involves longer synthesis-implementation cycles that may take

hours to complete, before a new program version can be finally deployed. This is at odds with current practices that foster continuous deployment cycles, and frequent updates to the packet processing programs [1, 7]. Finally, packet processing functions should consume only minimal FPGA hardware resources, to leave space to other accelerators required for signal processing, machine learning, etc.

These requirements, when combined, rule out existing solutions that cannot dynamically change the implemented packet processing programs, such as those based on the high-level synthesis of programs described with domain-specific languages [3, 38, 40] like P4 [8]. Alternative approaches, such as hXDP [10], explicitly address these challenges, but at the cost of a lower packet forwarding throughput. In fact, hXDP implements on the FPGA a processor-based executor for network programs described with eBPF, which provides a remarkable but still limited throughput performance, comparable to that of a single multi-GHz CPU's core [10].

Our goal is to improve on this figure, and significantly increase throughput while respecting the listed requirements. We follow the conceptual approach of hXDP, embracing the Linux's eBPF framework and its programming model to describe packet processing functions. However, we introduce a new technique, *program warping*, which leverages common properties of eBPF programs to automatically replace the execution of many program's instructions with a semantically equivalent hardware-supported implementation, thereby reducing the program execution time and increasing throughput. As we will see, in some real-world use cases our system can achieve up to an 18x higher throughput than hXDP.

Program warping builds on the observation that a subset of the eBPF programs' instructions implement common packet processing tasks, such as packet header parsing, which can be efficiently implemented in pipeline-parallel architectures [9, 13]. Therefore, under the constraint of keeping transparency to the programmer, we address two main issues in our design: (i) identifying, from the eBPF program's bytecode, the tasks that can be efficiently parallelized in a pipeline; (ii) designing an FPGA pipeline that runs such tasks, while providing runtime

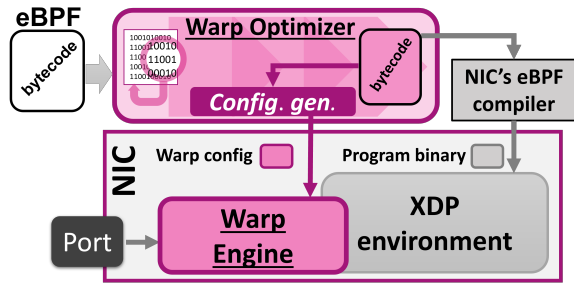


Figure 1: Program warping includes two components: an optimizing compiler and a hardware engine for FPGAs

reconfigurability and using minimal hardware resources.

We prototype program warping extending hXDP with a new compiler, the *Warp Optimizer*, and a new hardware module, the *Warp Engine* (cf. Figure 1). The Warp Optimizer performs static analyses of the eBPF bytecode, leveraging the eBPF’s machine model to make several simplifying assumptions, e.g., about memory areas’ content, in order to infer the program’s intent. In this step, the Warp Optimizer identifies set of instructions whose execution can be performed by the Warp Engine. The Warp Engine is integrated with the hXDP’s eBPF executor. Packets received by the system are processed by the Warp Engine first, and then passed to the eBPF executor to run the subset of the program that cannot be accelerated. In turn, this allows us to streamline the Warp Engine implementation, which resembles a fused parser plus match-action pipeline, supporting only the minimal set of functionality required to accelerate common packet processing tasks, and leaving to the eBPF executor any more complex functionality.

We evaluate our system with a 100Gbps Xilinx Alveo U50, running 6 real-world use cases: the IP router and the tunneling examples from the Linux XDP’s application examples; a Dynamic NAT; the Facebook load balancer Katran [14]; and the Network Security Appliance Suricata [39]. On this set of applications and compared to hXDP, program warping provides a per-application speed-up of at least 1.2-3x, and up to 18x, while running at 250MHz and keeping the overall FPGA resources occupation below 15%. To put this result in perspective, running Suricata in software (Linux v.5.4.0) on a single Intel Xeon 4410 CPU core achieves a throughput of about 8.7 Million packets per second (Mpps), whereas our program warping prototype achieves up to 83Mpps in a similar setting. This shows that program warping is beneficial in all cases, but clearer advantages appear with programs that have a heavier packet parsing and classification component. In summary, we contribute:

- A method to extract high-level packet processing tasks from eBPF bytecode, and generate functionally equivalent descriptions that combine parsing descriptions, match-action rules and subsets of the original bytecode;
- A hardware extension to hXDP, which implements high-performance and runtime-configurable packet data reading and classification using few FPGA resources;

- The evaluation of the end-to-end system using 6 real-world use cases and extensive micro-benchmarks.

2 Goal, Requirements and Challenges

Our goal is to provide a significant increase (i.e., >2x) to the throughput of eBPF packet processing programs on FPGA NICs, while meeting the following requirements:

1. The system should run unmodified eBPF programs
2. The system should support dynamic program loading
3. The system should use a small fraction of the FPGA resources (<20%)

This is challenging for several reasons. First, previous work like hXDP already explored the optimization space for eBPF programs on FPGA, leveraging instruction-set specialization and instruction-level parallelism to reduce the number of programs’ instructions and run them in parallel, when possible. This suggests that additional instruction-level optimization is unlikely to provide large gains. Second, our solution space is limited since we cannot change the eBPF programming model. For instance, we cannot pursue any solution that would require programmers to annotate their code, e.g., to discover parallelization opportunities. Third, the need to support dynamic program loading rules out approaches that implement programs as hardware pipelines, e.g., like in Emu [38] and P4->NetFPGA [3]. Finally, the requirement to use little FPGA resources makes effectively impossible to use any solution that requires complex logic implementation in hardware. For reference, even streamlined hardware designs like hXDP already consume about 10% of the FPGA resources.

Non-goal Since we use only a fraction of the FPGA for network packet processing, we do not have the goal of matching the throughput of designs entirely dedicated to the task.

3 Concept and Background

In this Section we provide background about eBPF/XDP, and then present the program warping concept and system design.

3.1 Background: eBPF and XDP

eBPF is a Linux technology used to implement load balancing [14], security [7], monitoring [2], deep packet inspection [5], policy enforcement [1], and more.

The eBPF framework runs small programs within the Linux kernel using a virtual machine (VM) with its own Instruction Set Architecture (ISA). The VM implements a register architecture, with a program counter (PC), 10 general registers ($R0 - R9$), and a read-only stack pointer ($R10$) that contains the address of a 512B memory area used as program’s stack. These capture the current program’s state, which resets for every new run. eBPF provides *maps* data structures to save state across program runs. These are memory areas defined at compile time and organized as lookup tables.

eBPF programs written in a high-level language, such as C, are compiled to the eBPF bytecode. The eBPF bytecode can be loaded in the kernel using different *hooks*. We focus on the XDP hook [19], and call *XDP programs* an eBPF program attached to the XDP hook. The hook is provided at the NIC driver level. When a packet is received, the XDP environment: (i) creates an `xdp_md` struct to contain the packet buffer pointers and metadata, such as the packet’s input port id; (ii) sets *R0* to point to the address of the memory area hosting the struct; (iii) and then starts the VM to run the XDP program. At the end of its execution, the program can return a forwarding decision for the packet by writing the forwarding action code in *R0*.

When loaded in the Linux kernel, the bytecode is statically verified to ensure safety, e.g., guaranteed program termination. To enable verification, eBPF programs can only use a subset of the C expressive power. For instance, unbounded cycles and dynamic memory allocations are not allowed. To finally run on the target hardware, a second (just-in-time) compilation step translates the eBPF bytecode to the target machine code. **Program example** Listing 1 shows an XDP program written in C. The program checks if the source MAC address of IPv4 packets is in a hashtable. If so, it passes the packet to the Linux’s network stack. Otherwise, the packet gets dropped. Furthermore, the program drops any IPv6 packet, and passes to the network stack any packet that is neither IPv4 or IPv6.

Listing 1: A simple eBPF/XDP program example in C

```

1 int l2_acl(struct xdp_md *ctx) {
2     void *data_end = (void *) (long) ctx->data_end;
3     void *data = (void *) (long) ctx->data;
4     void *lookup_res = NULL;
5     __u32 proto, nh_off;
6     struct ethhdr *eth = data;
7     __u8 key[6] = {0};
8     nh_off = sizeof(struct ethhdr);
9     if (data + nh_off > data_end) {
10        return XDP_DROP;
11    }
12    proto = eth->h_proto;
13    if (proto == BE_ETH_P_IP) {
14        __builtin_memcpy(key, eth->h_source, 6);
15        entry = bpf_map_lookup_elem(&map, &key);
16        if (entry) {
17            return XDP_PASS;
18        } else {
19            return XDP_DROP;
20        }
21    } else if (proto == BE_ETH_P_IPV6) {
22        return XDP_DROP;
23    } else {
24        return XDP_PASS;
25    }
26 }

```

3.2 Program Warping

eBPF executors on FPGA have limited throughput when they need to process many eBPF instructions: FPGA designs usually have a low clock frequency (e.g., <400 MHz), making running an instruction expensive. While parallel instructions execution is possible, the level of parallelization is at most 2-3 instructions per clock cycle [10]. Reducing the number of instructions can increase throughput, but *can we achieve functional equivalence with less instructions?*

To answer the question, we studied several XDP programs. We show two examples in Figure 2, where we report the control flow for the program from Listing 1 (a), and for (a subset of) Katran [14] (b), an XDP L4 load balancer deployed in production by Facebook. In all the studied cases, the first part of the program has instructions that only perform reading from the packet data and comparisons with constants. This is the case since network packet processing programs usually perform packet header parsing and classification as a first step. After that, the programs diverge significantly, with operations that are specific to the application logic.

For a hardware implementation, read only access to a single (small) memory means no data hazards to handle (i.e., no read/write conflicts), and that hardware wires routing may be simple, since a single memory contains all the needed data. In fact, the operations of the first program’s part may be described by a few match-action rules, as shown in Table 1

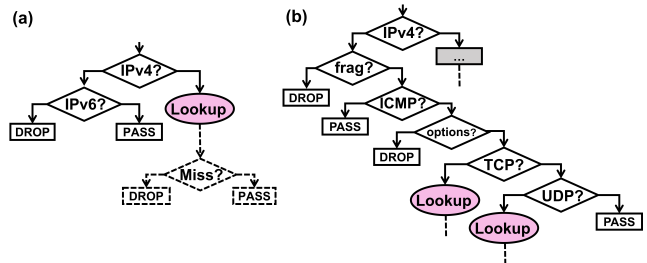


Figure 2: Flow diagrams for the program from Listing 1 (a) and for (part of) Katran (b). The solid lines show the part of the program that only needs reading packet data and comparison operations.

for Listing 1. Rules #1 and #3 only read some bits from the packet and check their value. Rule #2 is more complex, since it requires the execution of the program’s part that includes the lookup and its downstream operations. More generally, this second part of the program requires to read/write memory areas beyond that containing the packet data.

From this observation emerges the core idea of program warping: run the first program’s part on an extremely simple executor, and then use a downstream eBPF processor to run only the remaining, more complex instructions. This enables skipping several instructions, and increase throughput.

3.3 System Design

Without loss of generality, we design program warping as an extension to hXDP (Figure 1), the current state-of-the-art to

#	eth_proto	action
1	IPv6	DROP
2	IPv4	Continue Processing
3	*	PASS

Table 1: Match-action rules to implement part of the program from Listing 1. Rule #2 needs additional processing and accessing data beyond the packet’s content.

run XDP programs on FPGA NICs [10]. hXDP provides the XDP environment for the FPGA NIC, and a compiler that takes eBPF bytecode and outputs the machine code for the on-NIC XDP environment. We inherit the eBPF programming and deployment models from hXDP: from the perspective of an eBPF/XDP programmer, program warping does not introduce any change.

Compile time When loading an XDP program’s bytecode to the NIC, program warping extends the hXDP compiler by triggering a new compiler first: the *Warp Optimizer*. The Warp Optimizer performs static analysis on the bytecode to identify the instructions that can *warped*, i.e., they can be run by the Warp Engine. The output of this process is a configuration for the Warp Engine in the form of match-action rules.

Runtime At runtime, the Warp Engine receives packets first, and applies the match-action rules. If a forwarding decision can be already taken at this stage, the Warp Engine sets the value of the XDP Environment’s *R0*, and transfers the packet to the XDP Environment that carries out the forwarding action. If instead the program cannot run entirely in the Warp Engine, then a context restoration is triggered. Context restoration allows the eBPF executor to skip the warped instructions, while ensuring a correct internal state to start processing the remaining instructions. This involves copying data from the packet to the XDP Environment’s *R0 – R9* and stack memory, and setting the program counter to point to the next program’s instruction. The Warp Engine performs such operations in parallel while also transferring the packet to the XDP environment, where finally the processing continues to terminate the program’s execution. That is, the Warp Engine and the XDP Environment work in pipeline: while the XDP Environment processes a packet, the Warp Engine is processing the next packets. This ensures that the introduction of the Warp Engine *never* reduces the system throughput, and that in the worst case it only introduces an often negligible increase (10s of nanoseconds) of the packet processing latency.

4 Warp Optimizer

The Warp Optimizer is a custom compiler that takes as input the eBPF bytecode and produces as output: (i) the set of bits that should be extracted from the packet data; (ii) a set of match-action rules that will replace the *warped* (i.e., removed) program’s instruction; (iii) the description of the context associated to *context restore* actions. The rules’ match conditions are described by a set of couples (*offset*, *length*), which

specify the bits of the packet that should be read. The actions can be of one of the following two types:

- An XDP **forwarding decision** that neither modifies the packet nor the internal state of the system (e.g., the content of the maps), i.e., DROP, PASS, TX or REDIRECT;
- A **context restore** to continue execution in the XDP Environment, configured using the provided program counter and context (i.e., registers and stack content).

Here, there are two important design decisions that allow us to minimize hardware complexity. First, the definition of the match conditions may be thought as roughly corresponding to the definition of packet header’s fields, however the Warp Optimizer (and the Warp Engine) have no knowledge of what a header field is. We purposely avoided the implementation of a complete packet header parser logic [18], opting instead for a simpler set of reads of a sequence of bit vectors from the packet data. This allows us to avoid the implementation of state machines and enables a fully pipelined execution of the bit vector extraction. Second, the Warp Optimizer only provides a forwarding decision action when there is no modification to the packet and no *side effects* due to the packet processing, e.g., map accesses. Modification to the packet would require additional hardware machinery, e.g., to compute values and write them in the specific packet’s positions. Instead, accessing any internal state of the system would increase hardware complexity significantly, requiring a tighter integration with the XDP environment, and introducing potential data hazards due to e.g., read-after-write for packets processed back-to-back in the Warp Engine pipeline [37].

4.1 Program analysis

To extract the Warp Engine configuration, the Warp Optimizer performs static analysis of the input program. Here, recall that XDP programs can implement arbitrary computations, which generally complicates any static analysis task [29]. Nonetheless, eBPF is designed to simplify static verification of programs loaded in the Kernel, which helps also our analysis. In particular, we benefit from the definition of three logically distinct memory areas: (i) the packet buffer; (ii) the stack; (iii) and maps. Each of these areas can be easily identified. The packet buffer is retrieved from the `struct xdp_md`, whose address is in eBPF VM’s register *R1* when a program starts. The stack base address is stored in the read-only register *R10*. Finally, maps are always accessed using a specific eBPF helper function. With this information the Warp Optimizer can trace accesses to the different memory areas, and infer the evolution of the program state.

In greater detail, the Warp Optimizer first builds the program’s *Control Flow Graph* (CFG), e.g., see left part of Figure 3. The CFG is a directed graph, in which each *node* represents a code *block*, i.e., a set of instructions that are all executed if the program’s control flow triggers the execution of the block’s first instruction. The directed edges show how

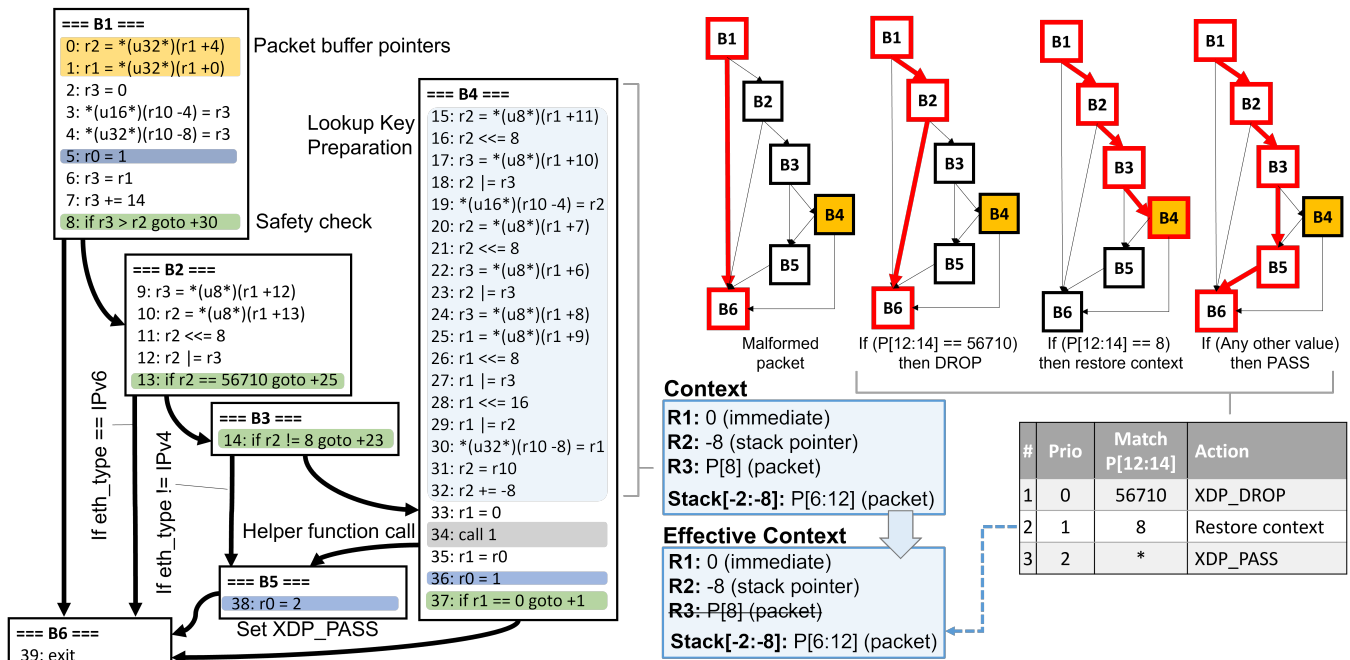


Figure 3: The operations of the Warp Optimizer for the program from Listing 1: (i) Control Flow Graph analysis; (ii) Match-action rules extraction; (iii) Context identification. All the operations are performed at compile time.

the different blocks might be executed one after the other, depending on the results of (conditional) jumps. Second, the Warp Optimizer converts the eBPF instructions, which use physical registers, into a Static Single Assignment (SSA) form. In this form, physical registers are substituted with variables that identify the instruction that have defined them. This helps the tracking of the values accessed by each instruction, and therefore it allows the Warp Optimizer to identify the accessed packet data, and the values stored into stack and registers.

After these two processing steps, the Warp Optimizer divides the CFG's blocks in three categories: start node; middle nodes; and terminal nodes. Terminal nodes are the blocks containing as last instruction `exit`, `call` or instructions that write to the packet data. Middle nodes are all the nodes that are not the start or terminal nodes, and they are further categorized in matching and non-matching nodes. This depends on whether the block ends with a conditional jump instruction (matching) or with any other instruction (non-matching).

4.2 Match-action rules generation

Match-action rules generated by the Warp Optimizer are triples $\langle matches, action, priority \rangle$, where *matches* is a list of (offset, length) pairs, *action* is either a forwarding decision or context restore, and *priority* is an integer value where the lower number encodes the higher priority. To generate these triples, the Warp Optimizer runs Algorithm 1. The algorithm defines a zero initialized current priority counter, a list of bitvectors extracted from the packet (*fields*), their corresponding matching values (*matches*), and the current stack and registers. Then, it performs Depth-First Search

(DFS) on the CFG, descending from the start node and stopping when it reaches a terminal node. The paths explored with this way capture the part of the program that can be *warped*.

When performing DFS, the algorithm evaluates all the instructions in the node, updating the current stack and registers state (i.e., the *registers* and *stack* arrays). For each middle node, if it is a matching node, the algorithm creates a copy of the *fields* and *matches*, and adds to them the bitvector checked by the current's block matching condition. That is, the condition of the conditional jump, and the variable's value used in the condition, respectively. This corresponds to checking `if packet_data[s:e] == X`, where `[s:e]` identifies a vector of $e - s$ bits in the packet starting at offset `s`, and `X` is an $e - s$ long bitvector. The current *registers* and *stack* are also copied, since the algorithm has to explore the two branches coming after the conditional jump, for which the program's state will evolve differently. When doing so, the algorithm explores first the branch corresponding to the *jump taken* case. When completing the exploration of that branch, the algorithm comes back to the latest encountered branching point, to explore the other branch, i.e., the one corresponding to the *jump not-taken* case (see right-top part of Figure 3).

A branch exploration terminates when there is a terminal node. After evaluating the instructions in the terminal node, the algorithm creates a match-action rule using the current list of *matches* and the current *priority* value. To define the action associated to the rule, the algorithm looks at the nodes's last instruction. If it is an `exit` instruction the action is a forwarding decision, defined by the currently evaluated value of the `r0` register. Other-

Algorithm 1: Warp Optimizer Algorithm

```
priority ← 0
matches, fields, rules, registers, stack ← []
Function get_MAT (block, matches, fields, priority, rules,
stack, registers) :
  evaluate_instructions (block, registers, stack)
  last_insn ← block.instructions[LAST]
  if is_terminal (block) then
    if is_exit (last_insn) then
      rule ← ⟨matches, Action (r0), priority⟩
    else
      action ←
        Action (PC=last_insn.pc, registers, stack)
      rule ← ⟨matches, action, priority⟩
    rules ← rules ∪ {rule}
    priority ++
  else
    block+ ← block.tnext
    block- ← block.fnext
    if is_match (last_insn) then
      fields ← fields ∪ {PacketField (last_insn)}
      matches+ ← matches ∪ {Match (last_insn)}
      get_MAT (block+, matches+, priority, rules,
stack, registers)
    get_MAT (block-, matches, priority, rules, stack,
registers)
```

wise, the action is a restore context action, which includes $\langle pc, restored_stack, restored_registers \rangle$, where pc is the program counter of the instruction immediately following the node's last instruction, $restored_stack$ and $restored_registers$ are the evaluated current stack and registers, i.e., the *context* to be restored (right-bottom part of Figure 3). After the rule creation, the priority counter is incremented. Since the CFG is explored by selecting first the branch-taken path, this ensures that the rules having the longer match list have higher priority, which is then useful to simplify the rule matching logic implementation at runtime.

5 Warp Engine

The Warp Engine is a pipelined implementation of a *fused* packet parsing and match-action unit, in principle similar to those implemented in switching ASICs, such as RMT [9], but with important conceptual differences and simplifications that are enabled by the co-design with the Warp Optimizer. For instance, in the Warp Engine there is no distinction between the input parser and the match-action unit.

Figure 4 shows an overview of the Warp Engine architecture. We can identify three conceptual sub-systems. First, there is a *key extraction unit*, which comprises twelve stages and is in charge of building a 16B long vector extracting bits from the packet data. Second, a *match-action unit* uses the key to perform a lookup for a matching entry, which is associated

with three areas in three distinct memories. These memory areas store the type of action associated with the packet, and the program context (register, stack) that should be restored in case of a context restore action. Finally, the last sub-system is the *context restoration unit*, which extracts the packet data required to build the context for a packet that needs to continue processing at the end of the Warp Engine. In our design, we use hXDP to implement the XDP environment on the FPGA, slightly modifying it to enable the Warp Engine to hook into the registers and stack memories.

An important aspect of the Warp Engine design is that the three sub-systems are part of a single pipeline that by design never stalls. In fact, the only case in which the pipeline stages do not advance processing is when hXDP is busy processing a previous packet, and therefore the hXDP's Active Packet Selector cannot host a new packet in its buffer memory. This design has also a second effect, since hXDP is still in charge of the forwarding of each and any packet, the Warp Engine has no impact on the packets ordering.

5.1 Key Extractor

The Key Extractor is connected to the packet input queue through a *splitter*, which duplicates the first 128B of the packet to forward them to the Key Extractor's pipeline. The pipeline includes 12 stages, and each of them implements a configurable extractor module. The extractor reads up to 2Bs from the duplicated packet chunk, performs a simple bitwise operation on them, e.g., and, with a 2B long constant value, and finally writes the result of such operation to a lookup key buffer. Which bytes to read, what operation to perform, and the value of the constant are all runtime-configurable parameters that are provided by the Warp Optimizer. Each extractor performs its operations in a single clock cycle, and passes to the next extractor: (i) its modified lookup key buffer; (ii) the offset at which to write in such buffer; (iii) the packet chunk.

5.2 Match-action Unit

The match-action includes a ternary-addressable content memory (TCAM), and three memory areas: (i) the *Action Memory*, to store the actions associated to the TCAM entries, including action type, $R0$ value and program counter; (ii) *Registers Configuration Memory*, which stores the Context Restoration Unit's configuration to extract the register values; (iii) *Stack Configuration Memory*, which provides a similar configuration but related to the stack. These three areas are organized in *lines* of different sizes, and for each memory the number of lines is equal to the maximum number of TCAM entries. The lookup key provided by the key extractor is used to find the matching entry in the TCAM, which is associated to a single *line number* that is then used to access in parallel the three memory areas. If the line extracted from the action's memory includes an action of type forwarding decision, then the pipeline propagates only such line to the next stage, to

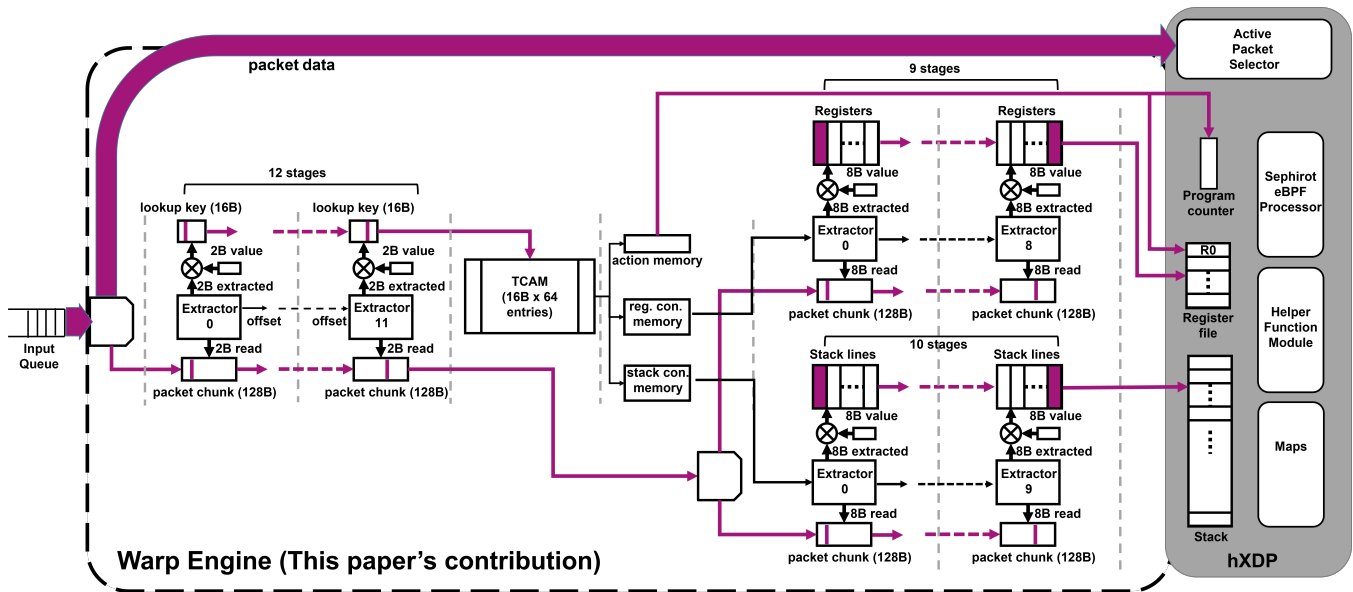


Figure 4: Warp Engine's architecture. The pipeline stalls only when hXDP is not ready to receive a next packet.

eventually configure the XDP Environment. Otherwise, each of the three lines extracted from the three memories is provided to the downstream pipeline. The memory lines contain the full configuration for the Context Restoration Unit, in order to build the stack and registers values. Both the TCAM entries and the memory lines are configured at runtime with the output of the Warp Optimizer.

5.3 Context Restoration Unit

The Context Restoration Unit comprises two parallel pipelines, which are composed of modules closely resembling the Extractor modules of the Key Extractor. However, instead of reading just up to 2B from the packet chunk, the extractors of the Context Restoration Unit can read up to 8B each. This is in line with the eBPF VM's registers size (64bit). Furthermore, they replace the lookup key buffer with larger buffers to host either the partially reconstructed stack or the reconstructed registers' state. Finally, instead of an offset, each extractor provides to its downstream extractor the line read from either the Registers Configuration Memory or the Stack Configuration Memory, depending on which of the two parallel pipelines the extractor belongs to. Here, we point out that this approach was needed since the Context Restoration Unit has an additional complexity element when compared to the Key Extractor. The Key Extractor configuration is the same for any received packet, whereas the configuration for the Context Restoration Unit strictly depends on the content of the received packet. Since the entire system is organized in a pipeline, each stage of the Context Restoration Unit has to carry along the configuration to restore the context for the specific packet being processed in that stage.

These two parallel pipelines are fed by a second *splitter* that duplicates the packet chunk. The pipeline that restores

the Stack has 10 stages, and it is connected to the Stack Configuration Memory. The line extracted from this memory provides to the Extractors the information needed to populate the stack, including constant values and values extracted from the packet chunk. This information includes: (i) the offset at which the packet chunk should be read; (ii) the operation to be performed on the read byte (and the constant value associated to that); (iii) the target address in the stack. The pipeline that restores the Registers has 10 stages too, including 9 Extractors and a Delay element. This is the case since only 9 registers need restoration ($R1 - R9$) and the Warp Optimizer ensures that $R0$ is only read if its value is changed by the loaded XDP program after restoration.¹ The Delay element is required to synchronize the two context restoration pipelines.

5.4 Integration with hXDP

The Warp Engine pipeline ends in hXDP. Here, the Warp Engine waits for hXDP to be available to receive packets. Once that is the case, it first copies the packet data in the hXDP's Active Packet Selector (APS). The packet data transfer is also pipelined, and it happens in synch with the Warp Engine's pipeline. That is, multiple packets' data are moved through the pipeline at each clock cycle, using a 64B datapath (matching Corundum's datapath). Then, we have two possible behaviors. If the current action memory's line contains a forwarding action, then the Warp Engine sets $R0$, and instructs the APS to proceed with packet forwarding. The APS will then carry out forwarding according to the value provided in $R0$. Instead, if the action is a restore context action, then the Warp Engine sets the hXDP's program counter, registers ($R1 - R9$)

¹Register $R10$ is read-only, and in hXDP it has a constant value. $R0$ is used to store the return value of helper function calls, which are usually the first instruction run by the program after restoration.

<i>Application</i>	<i>Instructions</i>		<i>TCAM Entries</i>	<i>Match size [B]</i>	<i>Max Stack size [B]</i>
	<i>eBPF</i>	<i>hXDP</i>			
L2 ACL	40	27	3	2	6
Router	119	95	9	4	8
Tunnel	283	155	7	4	24
DNAT	228	135	6	6	40
Suricata	138	65	49	12	40
Katran	1398	1013	20	16	80

Table 2: Tested applications and key metrics

and stack, and starts the hXDP’s Sefirot eBPF Processor. Sefirot will then run the XDP program starting from the instruction pointed by the program counter, and using the provided registers and stack values.

5.5 Implementation

We implemented the Warp Engine design using the latest version of hXDP, which is integrated in Corundum [16], clocked at 250MHz, and targets a Xilinx Alveo U50 FPGA NIC [4]. The Warp Engine is clocked at 250MHz too, and its pipeline is 28 clock cycles long. Since at 250MHz each clock cycle takes 4 nanoseconds, the Warp Engine introduces a fixed 112 nanoseconds of latency to each processed packet. This is a negligible overhead in the vast majority of cases, and it is the only runtime overhead introduced by the Warp Engine.

Our design has several parameters, e.g., the number of Key Extractor stages and the packet chunk size, which may be changed to meet different use case requirements. We summarize them in Appendix, along with the configuration we implemented in this paper, which is driven by the requirements of the 6 use cases we tested during evaluation (cf. Table 2).

6 Evaluation

In this Section we evaluate correctness, optimizations, resources requirements, and performance of our prototype.

6.1 Applications

We use 6 different applications to perform the evaluation, as detailed next. Table 2 summarizes them and reports relevant metrics, including their requirements in terms of Warp Engine’s TCAM entries, lookup key size and max Stack size. **L2 ACL (Running example)**. This is the application we used as running example, and described in Section 3.1.

Dynamic NAT. Network Address Translation (NAT) for flows coming from a LAN and destined to a public network, and reverse translation. The application has two main branches: (i) one for packets originated from the the LAN, and (ii) the other for those coming from the public network.

XDP Router. An implementation of an IPv4/IPv6 router, provided as eBPF application example with the Linux Kernel. It performs parsing of L2 and L3 headers, and then a lookup in two tables to take a packet routing decision.

XDP TX Tunnel. This is another eBPF application example provided by the Linux Kernel. It performs IPinIP encapsulation matching on destination IP address and destination L4 port. A lookup in a hashtable matches on the destination virtual IP address to retrieve the tunnelling information.

Suricata IDS. Suricata [39] is a software Intrusion Detection System (IDS). Among its multiple features, it provides an XDP program that works as a filter, to perform early dropping of undesired flows. The XDP program contains a large number of processing branches to handle all the combinations of stacked 802.1Q and 802.1AD VLAN headers, and performs a lookup in a hashmap to take some of the filtering decisions.

Katran. Katran [14] is an XDP-based Layer 4 load balancer. It encapsulates packets with a specific destination Virtual IP addresses and balances the connections towards the available servers. The first part of the processing includes L3 parsing and handling of ICMP/ICMPv6 protocols. Then, a first map lookup retrieves the virtual IP information. The application uses this information to query a Least Recently Used (LRU) map, in order to fetch the address of a connection table. A query to the connection table finally retrieves the real IP address of the destination server.

6.2 Functional Equivalence

Program warping modifies a program to run it on a system that comprises two different executors. We therefore performed tests to verify that the resulting behavior matches the original program behavior. In particular, for each of the tested applications, we: (i) enumerate all the program’s control paths; (ii) generate input packets that trigger the execution of each of the listed paths; (iii) and finally verify the produced output, for all the generated input packets. We run these steps for the 6 applications described earlier, verifying that program warping keeps functional equivalence. More details are in Appendix.

6.3 Warped instructions

We now evaluate the number of instructions that can be skipped thanks to program warping, since their functionality is implemented by the Warp Engine. This requires evaluating the instructions being actually executed at runtime. We use uBPF [22], a userspace eBPF processor, extending it to implement a Warp Engine emulator in software, to compute the number of actually executed instructions for all the tested applications, and for all the control flow paths of each application. Since the control path at later stages of the program depends also on the stored state, e.g., entries in the maps, our testing strategy is adapted to test the multiple possible state conditions. For instance, in the case of the L2 ACL, after the map lookup there are two different paths: if the lookup returns an entry; or not (cf. Listing 1).

Figure 5 reports the results, showing the total number of eBPF instructions executed per path (background bar), and the number of instructions executed when program warping is

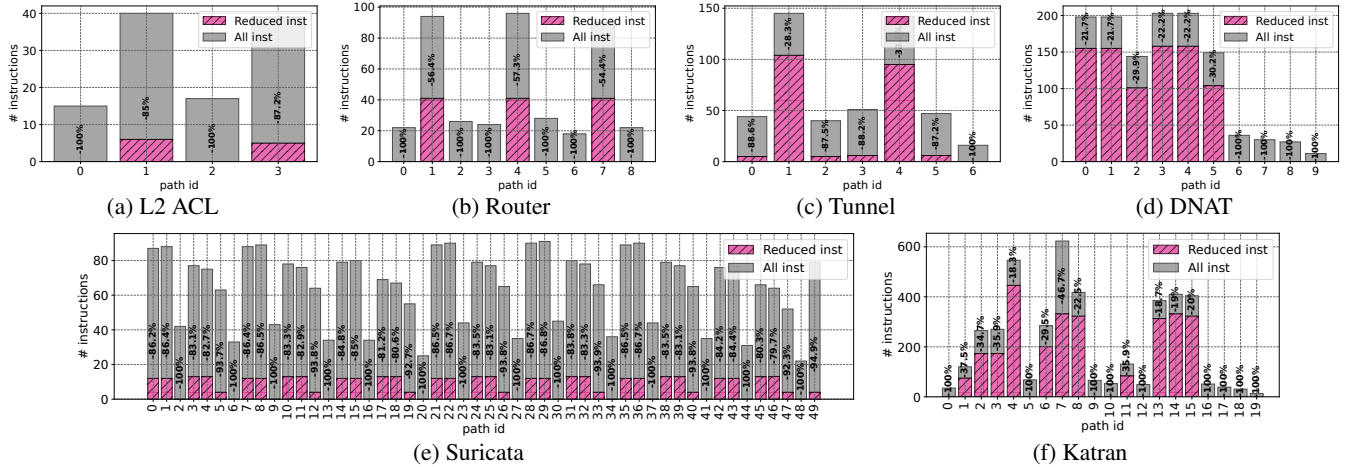


Figure 5: Number of instructions per program’s control flow path (background bar) vs number of instructions executed per path with program warping (foreground bar). Program warping reduces the instructions to be executed by 50-100%.

in place (foreground bar). The results show that in many cases the number of instructions to be executed by the eBPF processor is reduced by over 50%, and that anyway in all cases it is reduced by at least 16.3%. More precisely, an application’s control flow paths belong to one of two general categories, based on where their execution is going to be implemented: (i) *Mostly Warp Engine*; (ii) and *Mixed*.

Mostly Warp Engine In all the applications there are at least a few control paths whose processing is mostly implemented by the Warp Engine. This is due to the practice of including in the programs several checks to take an early forwarding decision. Many of these early decisions are taken to *protect* programs from bogus or malicious traffic, which has an interesting implication: program warping may provide higher performance boosts when it is the most needed. For instance, some Denial-of-Service attacks’ traffic may be entirely handled by the Warp Engine. Consistently, we can observe that all the control flow paths of Suricata fall in this category. In fact, Suricata generates XDP programs to filter network traffic as early as possible. As a result, in some cases the entire program is implemented by the Warp Engine. More generally, in Figure 5e we can see that the instructions reduction depends on the specific path, and it is in the range 82%-100%.

Mixed Some control flow paths split their execution between the Warp Engine and the eBPF processor, either in equal parts or mostly using the latter. This is the case for, e.g., Router, Tunnel, and DNAT. In such cases, the packet parsing and lookup key extraction are delegated to the Warp Engine. The rest of the application logic, e.g., lookup in a single map and packet header mangling, is performed by the eBPF processor. In some programs, this second part is relatively simple. For instance, we see that program warping reduces the instructions of these paths by 54%-57% and 51% for Router (Paths 1, 4, 7). In some applications, this second part is instead more complex. For instance, the Tunnel application’s paths 1 and 4 are reduced by 28% and 33%, respectively. This is the case

	Logic Res.		Memory Res.	
	LUTs	Reg.	BRAM	URAM
Corundum (C) + hXDP	10.7%	6.91%	13.65%	2.34%
C + hXDP + Warp Engine	16.8%	9.86%	14.51%	8.44%

Table 3: FPGA resources usage

since these paths have a large number of instructions that deal with the packet encapsulation, which happens after a map lookup. Similarly, in the first six DNAT application paths, the reduction ranges between 21%-31% due to the high number of instructions required to recompute the checksum and packet modifications after the program warping. In Katran’s paths this is even more evident due to the many lookups in maps performed in the program’s paths. The reduction, in this case, ranges from 16.8%-34.7%.

6.4 Warp Engine Hardware Requirements

We now evaluate the FPGA resources required by the Warp Engine. We compare the requirements with those of the latest hXDP version [4], which is integrated within the Corundum NIC [16] and targets a Xilinx Alveo U50. The U50 is equipped with a Xilinx Ultrascale+ FPGA, which offers 4 main types of resources that are of interest to us: (i) Lookup-Tables (LUTs); (ii) registers; (iii) block RAM (BRAM); and Ultra RAM (URAM). The LUTs and registers are the main building blocks to implement logic functions, whereas BRAM and URAM are two different memory blocks provided by the FPGA. BRAMs provide multi-port access, whereas URAM have a single read/write port but they are larger than BRAMs. For all the resource types, the Warp Engine is within our original requirement of keeping the packet processing subsystem below the 20% of the available FPGA resources (Table 3).

6.5 End-to-end performance

We finally test the end-to-end system when processing traffic, measuring both packet throughput and forwarding latency.

Testbed We use two machines: a first machine is equipped with a 100Gbps Mellanox ConnectX-5 NIC, and it runs a DPDK-based traffic generator/receiver, capable of sending traffic at 100Gbps with 64B packets, i.e., ~ 150 Mpps; the second machine is equipped with a single port 100Gbps Xilinx Alveo U50, connected back-to-back with the first machine. In all the tests, we measure packet forwarding that is handled entirely within the NIC, and drops. In this last case, we gain visibility by placing a dedicated drop counter within the FPGA design. Latency is always measured at the packet generator’s machine, as difference between the packet reception and packet sent (hw) timestamps. We do not measure the performance of processing that involves transferring packets to the host system, since the Corundum’s network driver can forward few Mpps, and it would therefore become the system’s bottleneck [16]. However, we remark that in terms of Warp Engine+hXDP design, the transmission to a NIC’s port or to the PCIe bus is implemented with the same hardware logic, therefore our system tests are representative of both cases.

Baseline We perform a baseline test to measure throughput and latency when 100% of program’s instruction are implemented by the Warp Engine. Since the Warp Engine pipeline performs the same steps for all the applications and execution paths, the performance is the same in all the cases. That is, we achieve ~ 83 Mpps, when performing DROP, and 50Mpps with a 1 μ s of per packet end-to-end latency when forwarding packets (TX). This is the same performance achieved by hXDP when running a program with a single instruction.² In fact, the Warp Engine relies on hXDP to carry out the forwarding action, and this performance matches the hXDP baseline performance, confirming that the Warp Engine is never a bottleneck in our design. In fact, this result holds true even when forwarding small packets that are larger than the Warp Engine packet datapath, e.g., 65B long packets.³

Applications Each application has multiple execution paths, which are taken depending on the received traffic and application’s state. For paths that are 100% processed by the Warp Engine, the performance is the same of the baseline case, for all applications and paths. This often provides throughput improvements of over 10x for such paths (E.g., see last row of Table 4). For the remaining paths, in the interest of space, we report the performance for only a subset of them, focusing on those that are the most frequent cases, or on cases that are interesting to study the system behavior. In particular, for L2 ACL, DNAT and Katran, we select the paths corresponding to successful lookups in the maps, which are the most frequent cases. For instance, this would be the path taken by established connections in both the DNAT and Katran cases. For Suricata, we see from Figure 5e a periodic pattern, which is due to the repetition of several different packet parsing

combinations (e.g., including or not multiple levels of VLAN parsing). Among these, we select the worst case for program warping, i.e., the path corresponding to the most instructions executed by hXDP. Finally, for Router and Tunnel, we analyze traffic traces to select the most common paths that would be triggered by processing such traffic. For Router, we use a Datacenter trace [6], and the path handling IPv4 is triggered in over 80% of the cases. For Tunnel, we use a MAWI trace [28], and the case IPv4+TCP is triggered in 60% of the cases.

We summarize the results in Table 4. For the selected execution paths, program warping improves throughput by 1.23x-3.08x, and increases latency in the worst case by only 104 *nanoseconds*. We can make two important observations. First, program warping provides remarkable throughput improvements, nonetheless, comparing to results from Figure 5, it seems that the tested paths provide a lower-than-expected speed-up. For instance, for the L2 ACL’s path #1, Figure 5a shows that only 15% of the instructions should be executed, which would suggest a potential throughput increase of over 6x. However, our test measures a 1.7x increase. This is the case since different hXDP instructions have different costs. For example, a `call` instruction may cost several clock cycles, and it also depends on variables such as the lookup key length. Therefore, the absolute number of instructions at compile time is not necessarily a good estimator for the achievable performance at runtime. Furthermore, it is important to notice that the Warp Optimizer works on the eBPF bytecode, which at a later stage is transformed by the hXDP compiler. The hXDP compiler may remove some instructions and parallelize others, therefore modifying the total program length (cf. Table 2). A side-effect of this is that the *warped* instructions may have been finally removed or parallelized by the hXDP compiler, which reduces the relative gain obtained by avoiding their execution. Second, in the case of Katran we observe a reversed result. Katran’s throughput is improved to 2.3x, despite Figure 5f shows only an 18.7% reduction for the path #11’s instructions. This is due to the relatively large number of (conditional) jumps in the first part of the Katran’s execution path. These jumps introduce bubbles in the hXDP’s processor pipeline, lowering throughput and increasing latency. In fact, Table 4 shows that in the case of Katran the Warp Engine significantly improves also forwarding latency, lowering it from 1.9 μ s to 1.5 μ s.

To put this in perspective, for the Router, Tunnel and DNAT cases the throughput of hXDP+Warp Engine (clocked at 250MHz) matches that of an Arm Cortex A72’s core clocked at 2.75GHz (A processor in use in high-end SmartNICs [32]). For Katran, the throughput is similar to that provided by two A72’s cores. In the case of L2_ACL and Suricata, our prototype matches the performance of four A72’s cores. More details about this are provided in Appendix.

²hXDP takes 3(5) clock cycles to handle drop(forward), with 64B packets.

³The interested reader can find more insights about the implications of datapath size on the packet forwarding throughput in [44].

Application [Path ID]	Latency [ns]		Tput [Mpps]		Tput w/WE vs w/o WE
	w/ WE	w/o WE	w/ WE	w/o WE	
L2 ACL [#1]	1128	1024	9.26	5.43	170.37%
Router [#7]	1304	1212	3.47	2.66	130.58%
Tunnel [#4]	1368	1288	2.84	2.21	128.41%
DNAT [#2]	1444	1364	2.34	1.89	123.36%
Suricata [#46]	1124	1112	10.86	3.52	308.52%
Katran [#11]	1501	1942	2.08	0.90	231.08%
<i>Performance for 100% instruction reduction scenarios</i>					
Suricata [#23] (DROP)			82.87	4.31	1824,72%

Table 4: Warp Engine (WE) End-to-End Performance

7 Discussion

Actual Performance Program warping throughput improvement depends on: (i) the XDP program; and (ii) on which program’s control path is executed. In our evaluations, we only measured a subset of the program’s paths. In operational settings, other control paths are likely to be part of the workload. In some cases, this may dramatically increase the performance gain. For instance, in Suricata, the last row of Table 4 shows the performance for one of the cases in which the Warp Engine can entirely offload hXDP. In this case, the system provides an 18.2x throughput increase. It is worth noticing that these paths are not necessarily uncommon or rarely executed. On the contrary, often they may represent the most frequently taken paths. For instance, CloudFlare defines XDP programs to perform early packet dropping for DDoS protection [7]. In such applications, these highly boosted paths are expected to be handling the majority of traffic.

Programming for Performance This last observation highlights that the performance of the loaded program is not guaranteed, and instead it depends on the received input. While this is sometimes considered an issue in switching devices [9], this is an expected behavior for software programmers who are already used to handle such variability in performance. A related interesting observation is that programmers can describe processing *rules* using `if` statements and *hardcoded* variables and constants, to improve throughput. This is the same set of techniques used to optimize XDP programs running within the Linux kernel on x86 processors. That is, program warping aligns to both the XDP programming model and best practices to improve program performance, matching XDP programmers’ expectations.

Configurability Our current program warping design is quite general, since the 6 presented applications include a good variety of cases. Nonetheless, there may be some other applications for which the Warp Engine resources cannot entirely describe the instructions that can be warped. While falling back to the eBPF executor is always a viable option, we also point out that it is possible to change several parameters of our design to accommodate different applications (e.g., lookup key’s size, TCAM entries number, etc.).

Limitations The performance acceleration provided by program warping strictly depends on the share of instructions that a program dedicates to packet parsing and classification. If the majority of the runtime is spent in other parts of a pro-

gram execution, program warping will only provide small benefits. Conversely, there could be cases in which the warp engine cannot offload all the program instructions that can be in principle *warped*. For example, this is the case if the Key Extractor’s pipeline is too short to extract all the data needed for parsing. In such cases, the Key Extractor and Context Restoration Unit’s pipelines length provides a hard limit to the maximum number of instructions that can be warped.

Scaling throughput The Warp Engine is not the system’s bottleneck. Therefore, for throughput oriented solutions where FPGA resources are available, it is possible to envision a design in which the Warp Engine serves packets to multiple hXDP modules that work in parallel. Similar high-throughput solutions is something we plan to explore as future work.

Portability While in this paper we use program warping to improve hXDP, the approach has more general applicability. In particular, the Warp Optimizer can be decoupled from the underlying hardware platform. For example, the extracted parsing logic can be used to automatically generate packet parsing programs specified with P4, or to map it to DPDK’s `rte_flow` API calls, to configure the underlying NIC packet parsing capabilities. Here, a challenge is to describe efficient mechanisms to move the partial *execution context* from e.g., the device subsystem performing header parsing and the subsystem that executes the remaining part of the program. Thus, the benefits of the approach vary depending on the specific target, which opens an interesting opportunity for future research. The Warp Engine design is also portable to different platforms, beyond FPGAs. In fact, it is a parametrized but “fixed” pipeline, thereby requiring relatively little changes to be ported to an ASIC implementation.

8 Related Work

A large number of new NIC designs appeared in the last few years [17, 20, 27, 32, 34, 41]. These solutions mostly combine in a mix-and-match manner different compute and network modules, e.g., regular NIC’s switching ASICs with general purpose compute clusters based on RISC cores [32], or FPGA-enhanced switching combined with general purpose clusters [20, 41]. In many of the solutions, a novelty factor is enabling P4-based programming of the switching ASIC. This effectively corresponds to replacing the fixed-function switching module with a programmable switching module [34]. However, in all these designs the data plane needs to be explicitly programmed with the provided tools, e.g., based on P4. Some of these designs offer (partial) eBPF support. However, they implement eBPF on top of the general purpose clusters, replicating the architecture commonly used in server machines, but on a smaller scale (including the need to transfer data from the switching ASIC to the general purpose compute clusters using an internal bus). In research, previous work addresses the challenges of moving data among these modules [25], and explores ways to

leverage these new NIC designs to improve application performance [12, 15, 24, 26, 35, 36, 43]. Program warping focuses specifically on the design of the packet switching module, targeting FPGA NICs, and presenting a solution that integrates with Linux applications that leverage eBPF/XDP. We extend hXDP [10], which to the best of our knowledge is the only solution providing full support for XDP on FPGA NIC directly within the switching module. Compared to hXDP, we provide better performance introducing a new compilation step co-designed with a hardware module, the Warp Engine, which is pipelined to the hXDP processor.

Recent work addressed eBPF programs optimization at compile time, targeting x86 processors [29, 42]. These works share with us the challenge of performing static analysis of the programs, and leverage some of the insights we discussed about the eBPF execution model. However, they focus on implementing compiler techniques targeting a fixed processor design, whereas we co-design the compiler and the hardware executor. Another related work is Gallium [43], which targets the offloading of a program's part to programmable switching ASICs. Also in this case, it assumes a fixed set of executors, including programmable switching chips and processors. Program warping, instead, introduces both a compiler and hardware design that integrates with the XDP processor, in order to push the intermediate computations context directly within the processor environment.

9 Conclusion

We introduced program warping, a method that leverages compiler-hardware co-design to accelerate the execution of eBPF programs running on FPGA NICs. Program warping enhances existing systems that run eBPF on FPGA NICs with a new compilation step and adding a hardware module, the Warp Engine. The compilation step identifies parts of eBPF programs that can be more efficiently implemented by the Warp Engine, which offloads them from eBPF processors, improving throughput (120%-300%, and up to 18x) at the cost of a small amount of additional FPGA resources. The crucial insight is that only packet data reads and comparisons are needed to implement the identified program parts. Therefore, the Warp Engine supports this minimal set of operations, thereby achieving speed and efficiency, while eventually concluding packet processing on a regular eBPF processor that could handle any more complex program's functions.

Acknowledgements

We thank the anonymous USENIX ATC 2022 shepherd and reviewers for their valuable feedback. This work has been partially funded by the European Commission in the frame of the Horizon 2020 projects 5GMED (grant #951947) and MARSAL (grant #101017171).

References

- [1] Cilium website. <https://cilium.io>.
- [2] Hubble github repository. <https://github.com/cilium/hubble>.
- [3] P4-NetFPGA. <https://github.com/NetFPGA/P4-NetFPGA-public/wiki>.
- [4] Pushing xdp into smartnics. https://fosdem.org/2021/schedule/event/sdn_hxdp_fpga/.
- [5] Suricata Documentation. Using Capture Hardware: eBPF and XDP. <https://suricata.readthedocs.io/en/latest/capture-hardware/ebpf-xdp.html>.
- [6] T. Benson. Data set for IMC 2010 data center measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html.
- [7] G. Bertin. Xdp in practice: integrating xdp into our dds mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, volume 2, 2017.
- [8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.
- [9] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM '13*, page 99–110, New York, NY, USA, 2013. Association for Computing Machinery.
- [10] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco. hxdp: Efficient software packet processing on FPGA nics. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990. USENIX Association, Nov. 2020.
- [11] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.
- [12] D. Chiou. The microsoft catapult project. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 124–124. IEEE, 2017.

- [13] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargafik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17*, pages 1–14, New York, NY, USA, 2017. ACM.
- [14] Facebook. Katran source code repository. <https://github.com/facebookincubator/katran>, 2018.
- [15] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, Apr. 2018. USENIX Association.
- [16] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen. Corundum: An open-source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2020.
- [17] Fungible, Inc. S1 DPU Product Brief. <https://www.fungible.com/wp-content/uploads/2021/01/PB0029.01.12020113-Fungible-S1-Data-Processing-Unit.pdf>.
- [18] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS '13*, page 13–24. IEEE Press, 2013.
- [19] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies, CoNEXT '18*, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.
- [20] Intel Corporation. Infrastructure Processing Units (IPUs). <https://www.intel.com/content/www/us/en/products/network-io/smartnic.html>.
- [21] Intel Corporation. 5G Wireless. <https://www.intel.com/content/www/us/en/communications/products/programmable/applications/baseband.html>, 2020.
- [22] IOVisor Project. uBPF repository. <https://github.com/iovisor/ubpf>.
- [23] D. Korolija, T. Roscoe, and G. Alonso. Do OS abstractions make sense on fpgas? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 991–1010. USENIX Association, Nov. 2020.
- [24] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16*, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery.
- [25] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259. USENIX Association, Nov. 2020.
- [26] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM '19*, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.
- [27] Marvell Technology, Inc. Data Processing Units. <https://www.marvell.com/products/data-processing-units.html>.
- [28] MAWI. MAWILab traffic trace - samplepoint f - 2021-03-22. <https://mawi.wide.ad.jp/mawi/samplepoint-F/2021/202103221400.html>.
- [29] S. Miano, A. Sanaee, F. Risso, G. Rétvári, and G. Antichi. Dynamic recompilation of software network services with morpheus, 2021.
- [30] NEC. Building an Open vRAN Ecosystem White Paper. <https://www.nec.com/en/global/solutions/5g/index.html>, 2020.
- [31] Netronome. AgilioTM CX 2x40GbE intelligent server adapter. https://www.netronome.com/media/redactor_files/PB_Agilio_CX_2x40GbE.pdf.
- [32] NVIDIA Corporation. NVIDIA BlueField data processing unit (DPU). <https://www.nvidia.com/en-us/networking/products/data-processing-unit/>.
- [33] Orange. OKO. <https://github.com/Orange-OpenSource/oko>.

- [34] Pensando Systems. Pensando DSC-100 Product Brief. <https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-100-Product-Brief.pdf>.
- [35] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: A programming system for nic-accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, Carlsbad, CA, Oct. 2018. USENIX Association.
- [36] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano. Flowblaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, Boston, MA, Feb. 2019. USENIX Association.
- [37] A. Sivaraman, A. Cheung, M. Budi, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM '16*, ACM SIGCOMM '16, pages 15–28. ACM, 2016.
- [38] N. Sultana, S. Galea, D. Greaves, M. Wojcik, J. Shipton, R. Clegg, L. Mai, P. Bressana, R. Soulé, R. Mortier, P. Costa, P. Pietzuch, J. Crowcroft, A. W. Moore, and N. Zilberman. Emu: Rapid prototyping of networking services. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 459–471, Santa Clara, CA, July 2017. USENIX Association.
- [39] Suricata. Suricata IDS Website. <https://suricata.io/>.
- [40] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research, SOSR '17*, page 122–135, New York, NY, USA, 2017. Association for Computing Machinery.
- [41] Xilinx, Inc. Alveo SN1000 SmartNIC. <https://www.xilinx.com/applications/data-center/network-acceleration/alveo-sn1000.html>.
- [42] Q. Xu, M. D. Wong, T. Wagle, S. Narayana, and A. Sivaraman. Synthesizing safe and efficient kernel extensions for packet processing. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference, SIGCOMM '21*, page 50–64, New York, NY, USA, 2021. Association for Computing Machinery.
- [43] K. Zhang, D. Zhuo, and A. Krishnamurthy. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20*, page 283–295, New York, NY, USA, 2020. Association for Computing Machinery.
- [44] N. Zilberman, G. Bracha, and G. Schzukin. Stardust: Divide and conquer in the data center network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 141–160, Boston, MA, Feb. 2019. USENIX Association.

Appendices

A Warp Engine Parameters

In Table 5 we report a subset of the parameters that can be configured in the Warp Engine. This is helpful to accommodate different workloads, or to tune the FPGA resources requirements to the workload of interest in the specific deployment. For instance, there may be cases in which a packet chunk of 64B is sufficient to extract the entire packet context. Likewise, there may be needs to extend the lookup key beyond 16B, etc.

Parameter	Val.	Description
Packet chunk size	128B	Number of packet's bytes that can be read to build the lookup key (also affects the TCAM entries width).
Lookup key size	16B	Size of the lookup key used in the match-action unit
Key Extractor Stages	12	Corresponds to the maximum number of different places that can be read in the packet chunk
Key Extractor read size	2B	Maximum number of contiguous bytes that can be read by each Key Extractor's stage (also affects the maximum size of the constant value used for the bitwise operation)
TCAM entries	64	Maximum number of match-action entries that can be configured by the Warp Optimizer for a given program
Stack Extractor Stages	10	Corresponds to the maximum number of different places that can be read in the packet chunk
Stack buffer	136B	Maximum number of stack bytes that can be restored
Stack Extractor read size	8B	Maximum number of contiguous bytes that can be read by each Stack Extractor's stage (also affects the maximum size of the constant value used for the bitwise operation)
Reg. Extractor Stages	9	Corresponds to the maximum number of different places that can be read in the packet chunk
Reg. Extractor read size	8B	Maximum number of contiguous bytes that can be read by each Stack Extractor's stage (also affects the maximum size of the constant value used for the bitwise operation)

Table 5: Warp Engine's main design parameters, and the values used for the design tested in this paper.

B Applications

Here we report a slightly more detailed description of the application used during our system evaluation (and reported in the paper).

L2 ACL (Running example). This is the application we used as running example, and described in Section 3.1. It includes three branches: the main processing branch handles IPv4 packets and checks whether the source MAC address is present in the access list; the other two branches handle IPv6

packets, which are always dropped), and any packet that is not IP, which is passed to the networking stack.

Dynamic NAT. Network Address Translation (NAT) for flows coming from a LAN and destined to a public network, and reverse translation. The application has two main branches: (i) one for packets originated from the the LAN, and (ii) the other for those coming from the public network. When a flow's first packet from the LAN is processed, the application selects a new *NATed* port, and saves it in the NAT binding table using the 5-tuple as flow identifier. Then it performs address translation and forwards the packet. For any following flow's packet, the application retrieves the NATed port, and performs address translation accordingly. In a similar way, packets from the public network are subject to a reverse NAT if there is a corresponding entry in the NAT binding table, or they are dropped otherwise.

XDP Router. An implementation of an IPv4/IPv6 router, provided as eBPF application example with the Linux Kernel. It performs parsing of L2 and L3 headers, and then a lookup in two tables to take a packet routing decision. The first table is an exact match table that looks up the entire IP destination address. If the lookup in the first table fails, the application performs a second lookup in a Longest Prefix Match (LPM) table.

XDP TX Tunnel. This is another eBPF application example provided by the Linux Kernel. It performs IPinIP encapsulation matching on destination IP address and destination L4 port. The application works with both IPv4 and IPv6, with the two main processing branches handling these two cases to assign the proper IPv4 or IPv6 encapsulation. A lookup in a hashtable matches on the destination virtual IP address to retrieve the tunnelling information.

Suricata IDS. Suricata [39] is a software Intrusion Detection System (IDS). Among its multiple features, it provides an XDP program that works as a filter, to perform early dropping of undesired flows. The XDP program contains a large number of processing branches to handle all the combinations of stacked 802.1Q and 802.1AD VLAN headers. After VLAN parsing, it processes differently IPv4 and IPv6 packets, and performs a lookup in a hashmap providing the 5-tuple plus the (optional) VLAN identifiers to take some of the filtering decisions.

Katran. Katran [14] is an XDP-based Layer 4 load balancer. It encapsulates packets with a specific destination Virtual IP addresses and balances the connections towards the available servers. The first part of the processing includes L3 parsing and handling of ICMP/ICMPv6 protocols, for early response to echo request messages. Then, a first map lookup retrieves the virtual IP information. The application uses this information to query a Least Recently Used (LRU) map, in order to fetch the address of a connection table. A query to the connection table finally retrieves the real IP address of the destination server. If a destination is not found, and the packet has the SYN flag set, then Katran installs a new forwarding rule in

the connection table to ensure forwarding consistency for the following packets of that flow.

C Software Emulator

For some of the Warp Optimizer evaluations, we used a Warp Engine emulator based on `uBPF` [22]. Here we give additional details about such implementation.

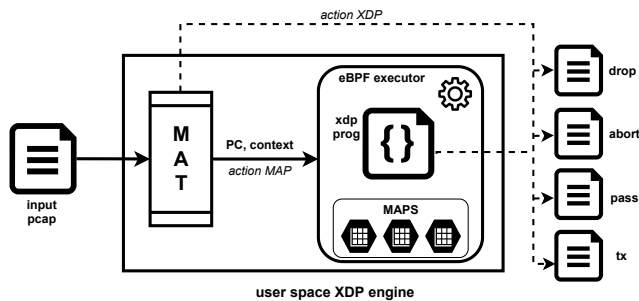


Figure 6: MAT+uBPF Architecture

`uBPF` is an open source project that implements an eBPF processor in userspace. Unfortunately, out of the box, `uBPF` misses relevant functional blocks, such as maps. We therefore used OKO [33], an open source project providing an eBPF engine for OpenVSwitch, to enhance `uBPF` with the OKO’s maps and helper functions implementations. Starting from this basis, we further extended this implementation to include any additional feature required by the Warp Engine (cf Figure 6).

Our implementation takes several input files to configure its internal modules. First, we implemented a program that reads the ELF files provided by the standard LLVM eBPF compiler, and which creates as output:

1. a text file containing the *eBPF instructions*, formatted as a sequence of bytes representing the 64bit instructions of the program;
2. a JSON file describing the XDP program’s map definitions. Such information includes the key size, value size, number of entries and type of map (array, hashmap, etc.).

These two files are provided to our software emulator to configure the eBPF executor, and create the maps required by the program.

The emulator’s Warp Engine module implements a Match Action Table (MAT) with ternary match values. The MAT is configured using the output generated by the Warp Optimizer. **uBPF execution** The emulator takes packet in input by reading a PCAP file. It can then be run in two different modes of operation: with the software Warp Engine disabled; and with the software Warp Engine enabled.

In the first case, the MAT is bypassed and the packet is directly fed to the eBPF executor, which applies the eBPF instructions on the packet data. With the MAT enabled, instead, the packet data is used to extract the fields needed to perform

a lookup in the MAT. The matched entry contains the action that must be performed on the packet, that is:

- a standard XDP return code (DROP, ABORT, PASS and TX), so in this case the packet bypasses the eBPF execution stage;
- a context restoration action, which contains the information to construct and restore the context.

In case of context restoration, we copy the registers and stack values as described in the action (which is configured from the Warp Engine’s output). Then, in any case, the program counter is updated with the one required by the matched action, and the regular eBPF execution starts. Finally, the emulator outputs a number of global and per-packet statistics:

- one PCAP trace for each XDP return code, for packets that have been subject to the DROP, PASS, ABORT and TX XDP actions;
- the list of instructions executed for each packet;
- the number of instructions actually executed;
- the number of times a rule in the Match Action Table has been matched.

These statistics have been collected to construct the results shown in Section 6.

D Functional Equivalence

We provide more details about the strategy we used to check functional equivalence of programs running with and without program working. In particular, we validate the equivalence of running an eBPF program in our accelerated system and running the same program in the standard Linux kernel XDP implementation. First, we analyze the *behavioral equivalence*, i.e. that the packets out of the Linux kernel implementation exactly match the packets in output from our software implementation. This is a black-box test and has two outcomes: (i) it validates the equivalence between the standard implementation and our accelerated version, and (ii) it validates the correctness of our software prototype. For what concerns the test cases, for each application we use synthetic packet traces in which each packet exactly matches one entry in the Match Action Table. We run an eBPF program with and without the MAT enabled and compare the output packet traces. We obtain the behavioral equivalence by verifying that the two outputs match exactly, in terms of packet data and associated XDP action.

Nonetheless, a careful choice of the test cases should take into account all the possible inputs such that the totality of the eBPF instructions of a program are covered. For example, in the NAT application, the first packet of a new connection matches the same entry of subsequent packets, but for the first packet the processing is different, and the instructions covered are different as well. In other words, we should take into account the state updates in the execution of a program. To validate the correctness of our test cases to cover the entire set of program’s instructions, for each application we crafted

the packets to cover all the branches in the Control Flow Graph, along with the correct configuration of the eBPF map entries. For every use case considered, we achieved the full program instructions coverage. We verified this by checking that the enumerated instructions represent the totality of the original program.

In the case of Katran, we used a simplified version by removing some parts of the code for which the instructions could not be executed. For example, some portions of the Katran code rely on timeouts triggering a certain condition. Since our uBPF prototype does not implement timers, we removed those program parts. In any case, all the instructions that are not covered by our tests always happen after the *warped* part of the program, therefore we could still verify that program warping does not modify in any way a program's behavior.

E Comparison with commercial SmartNICs

In Section 6 we compare our prototype only with hXDP. This is the case since we are not aware of any other NIC platform that supports running unmodified eBPF in the network data plane.⁴ Furthermore, we are interested in evaluating the specific contribution of program warping to FPGA-based eBPF executors, and less concerned with the evaluation of packet processing when using different platforms. However, when looking more generally supporting eBPF on a NIC, some recent commercial NICs that include battery of general purpose CPUs can indeed run unmodified XDP programs.

It should be clear that comparing our program warping prototype with such systems is not generally correct from a technical perspective, since the goals, constraints and scopes of application are too different to devise a fair testing strategy. For instance, a more direct comparison of the packet forwarding data plane component would require an ASIC-based implementation of program warping.

While we are aware of the above, we believe that the comparison may still be useful for practitioners who may be interested in evaluating FPGA NIC solutions vs alternatives. Therefore we decided to at least include such tests in this appendix for the interested reader.

NVIDIA Bluefield2 architecture We tested the XDP programs performance on an NVIDIA Bluefield2 NIC [32] (in NVIDIA terminology, these devices are currently called Data Processing Units, or DPUs). The Bluefield2 combines two main subsystems: a switching data plane based on the Mellanox ConnectX6 architecture; and a battery of 8 general purpose Arm A72 CPUs running at up to 2.75GHz. The ConnectX6 receives the packets from the network ports and can forward them directly to the host system, like a regular NIC,

⁴In fact, Netronome SmartNICs [31] support eBPF, but only in a limited form, and therefore packet processing programs need to be rewritten for the specific Netronome's capabilities.

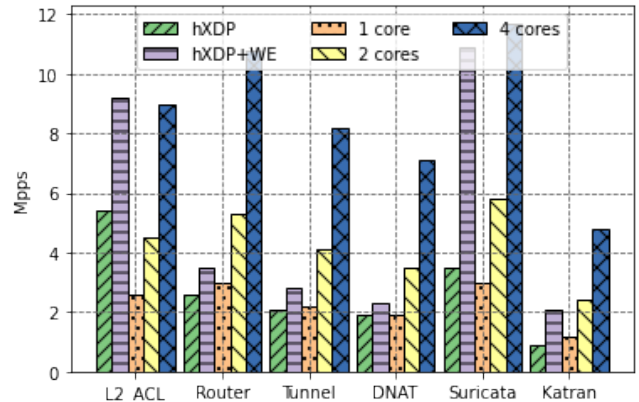


Figure 7: Forwarding throughput for the applications described in Section 6, when running on hXDP, hXDP+Warp Engine, and on 1-4 cores of an NVIDIA Bluefield2's CPUs.

or it can re-direct them to the Arm CPUs. Here, further processing can happen, and the packets can be either *consumed* locally, or sent back once more to the ConnectX6, to be finally delivered to the host system or to the network port.

Experiments We use the experimental setup, applications and testing strategy described in Section 6 to evaluate the packet forwarding performance of the Bluefield2, when using the Arm CPUs. Figure 7 shows the results.

Like already explained in Section 6, we report the results only for a subset of the application paths. For Router, Tunnel and DNAT, the hXDP+Warp Engine combination achieves a higher throughput than an Arm core clocked at over 10x the hXDP clock frequency. For Katran, our prototype is close to the performance provided by two cores. Finally, in the case of L2_ACL and Suricata, the hXDP+Warp Engine achieves a forwarding throughput roughly equivalent (or close) to 4 Arm cores instead.

These results show that in favorable cases program warping can indeed boost the performance of an FPGA-based processor to match that of several hardcoded cores running at much higher frequency. This suggests that a practitioner will require a careful workload analysis if the choice of an FPGA NIC is not mandated by other deployment requirements⁵, since the performance is use case dependent. In any case, program warping provides a viable solutions to run eBPF software packet processing in environments where an FPGA NIC is required.

⁵Requirements may not be necessarily related to the need of hosting FPGA-based accelerators. They may also include considerations on power consumption and type of board cooling.



NVMe SSD Failures in the Field: the Fail-Stop and the Fail-Slow

Ruiming Lu^{1*}, Erci Xu^{2*}, Yiming Zhang^{3†}, Zhaosheng Zhu⁴, Mengtian Wang⁴,
Zongpeng Zhu⁴, Guangtao Xue^{1†}, Minglu Li^{1,5}, and Jiesheng Wu⁴

¹Shanghai Jiao Tong University, ²PDL, ³Xiamen University,
⁴Alibaba Inc., and ⁵Zhejiang Normal University

Abstract

NVMe SSD has become a staple in modern datacenters thanks to its high throughput and ultra-low latency. Despite its popularity, the reliability of NVMe SSD under mass deployment remains unknown. In this paper, we collect logs from over one million NVMe SSDs deployed at Alibaba, and conduct extensive analysis. From the study, we identify a series of major reliability changes in NVMe SSD. On the good side, NVMe SSD becomes more resilient to early failures and variances of access patterns. On the bad side, NVMe SSD becomes more vulnerable to complicated correlated failures. More importantly, we discover that the ultra-low latency nature makes NVMe SSD much more likely to be impacted by fail-slow failures.

1 Introduction

NVMe SSD is now the new favorite of modern data centers. With a performance specification of up to 6GB/s bandwidth and microsecond-level latency, NVMe SSD serves as a strong performance upgrade to its SATA-based peers [8, 18, 29–31].

Apart from the performance, the reliability of any hardware under mass deployment is of great concern [3, 5–7, 10, 14, 38, 40, 42, 45]. While there is a spate of work covering the failure characteristics of SATA SSDs in the field [34–36, 41, 47], their findings may not be conclusive for NVMe SSD.

First, with a low-latency interface, NVMe SSD can be especially *prone* to fail-slow failure (aka. gray failure [17, 21, 25, 26, 48]). In a nutshell, the NVMe SSD fail-slow failure causes a drive to exhibit abnormal performance slowdown (e.g., high latency under normal traffic). Unlike SATA SSD, where fail-slow failure may be masked by the relatively high latency (>100μs), NVMe SSD can be easily impacted due to its ultra-low latency nature (~10μs) [23, 27, 28].

Moreover, the NVMe SSD is not just the SATA SSD with an interface upgrade. Instead, the internal architecture of NVMe SSD has gone through considerable changes. An outstanding example is the wide adoption of 3D-TLC NAND in NVMe SSD for larger capacity. Compared to MLC, the denser bits per cell (i.e., TLC) shows lower reliability and

the vertical stacking (i.e., 3D flash) can exhibit disparate behaviors or even opposite patterns (e.g., lower error rate under higher temperatures [32]). Also, the vendors have integrated a series of techniques to improve the overall reliability in NVMe SSD, such as Redundant Array of Independent NAND (RAIN) or Low-Density Parity-Check code (LDPC) [43, 50]. Unfortunately, with no large-scale NVMe SSD fail-stop study available at the moment, the influences of recent advancements remain unknown.

In this paper, we study the fail-stop and fail-slow failures of NVMe SSDs deployed at Alibaba. Specifically, we collect and analyze device logs (i.e., SMART [11]), runtime logs (i.e., *iostat*), and failure tickets from over one million NVMe SSDs¹. Throughout the study, we set our analysis into the context of previous studies to help various parties of interest get a clear picture of NVMe SSD reliability, including the improving and deteriorating failure patterns of fail-stop failures and the characteristics regarding the fail-slow failures.

We start our study by plotting and analyzing the baseline statistics (§3) of the NVMe SSDs, including the drive characteristics (e.g., manufacturer and model), usage characteristics (e.g., power-on time), and health metrics (e.g., annual replacement rate). Then, we comb through the dataset against different impact factors such as age and write amplification (§4). Finally, we lay a special focus on the fail-slow failures (§5), where we rigorously identify the fail-slow drives and perform extensive analysis. Altogether, we obtain 10 major findings and we list the highlights as follows:

- Infant mortality (failures occurring soon after deployment), a concerning failure trend in SATA SSD [35], is not outstanding in NVMe SSD. For nearly all of our models, the failure rate in the first three months is equivalent to or even less than that from later periods.
- High Write Amplification Factor (WAF), unlike SATA SSD [36], is no longer closely correlated with failures. Interestingly, NVMe SSD with low WAF (WAF≤1) exhibits 2.19× higher ARR than high-WAF ones.
- Co-located (i.e., intra-node/rack) NVMe SSD failure becomes more temporally correlated. For example, compared to SATA SSD, NVMe SSD correlated failure increases up

*Equal contribution.

†Corresponding authors.

¹We release our dataset at <https://tianchi.aliyun.com/dataset/dataDetail?dataId=128972>.

to $14.69\times$ and $1.78\times$ in intra-node/rack scenarios, respectively.

- The fail-slow failure is a widespread and severe problem for NVMe SSD. On average, 1.41% of NVMe SSDs are infected within four-month monitoring, which is $6.05\times$ that of HDD. Besides, fail-slow NVMe SSD could degrade to SATA SSD or even HDD performance.
- The NVMe SSD fail-slow failure does not correlate with SMART attributes, and rarely (0.22% of the fail-slow drives) transits to fail-stop failures.

We conclude this paper with the limitation of this study (§6), the related work (§7) and a short conclusion (§8).

2 Background

2.1 System Architecture

The NVMe SSDs, in our study, come from multiple IDCs (Internet Data Centers) across the globe. An IDC usually hosts dozens of storage clusters. The clusters are homomorphic with each running an HDFS-like distributed file system (DFS). Each cluster owns several to tens of racks and each rack includes up to 48 nodes. All NVMe SSDs in our study come from the all-flash-configuration nodes that contain 12 NVMe SSDs (not RAIDed) for data storage (not hosting OS).

2.2 Drive Model & Workload

Our candidate SSDs are all enterprise-level. The earliest model was deployed around May 2015, while the latest model is from July 2019. We introduce the details in §3.

The SSD fleet serves a total of 7 services, including block storage, object storage, big data, buffering, log, streaming, and query. Each service spans across several dedicated clusters. For confidentiality, we do not share the numbers of SSDs or their distribution under each service. Still, we study the influences of workload under controlled-variable experiments.

2.3 Data Collection

Data	Span	Entry
SMART Logs	2019-11-04~2020-11-14	~1.8M
Perf. Logs	2020-11-16~2021-03-05	~84M
Failure Tickets	2019-11-04~2020-11-02	~20K

Table 1: Data collection period (§2.3).

SMART logs. SMART is a set of attributes widely adopted by vendors and administrators to evaluate the reliability and performance of drives [11]. In our clusters, the reportings of SMART attributes are collected on a daily basis. Readings of the metrics can be either cumulative (e.g., number of media errors) or instantaneous (e.g., temperature). In practice, vendors may not necessarily follow the exact counting or reporting mechanism. Therefore, we standardize the numbers based on the manufacturer manuals.

Performance logs. A major subset of our clusters is equipped with node-level daemons to monitor and record the `iostat`, a Linux kernel performance log. The `iostat` includes vital statistics of storage devices, such as latency, IOPS and throughput. Currently, the daemon runs 3 hours a day (from 9 PM to 12 AM) and only records the average `iostat` values of each monitoring window (15 seconds long). Within the three hours, the traffic is relatively stable (around 70% peak traffic) and dominated mainly by internal workloads and large external clients (i.e., less burst traffic).

Failure tickets. Every node in our clusters has set up a daemon to monitor and report fail-stop failures. Upon reporting, a failure ticket would be generated (and manually checked by engineers), containing basic information of the victim drive (e.g., model and hostname) and the timestamp. Around 35% of our nodes also record an error code, detailing the direct symptom of failure (see Table 3 in §3.1). Upon failures, based on the symptoms, drives would be repaired online (e.g., `fsck`) or directly put offline for replacement (e.g., drive lost).

2.4 Methodology Correctness

To ensure sound and generalizable conclusions, we adhere to the following principles and measures throughout the study.

First, our study methodology (similar to [19, 34, 36, 41, 47]) starts with a general and extensive comparison to identify outstanding dominant factors. If high-level observation is fruitless or suspicious (e.g., spurious correlation led by interdependence between different factors as noted in [41]), we would then perform fine-grained controlled variable experiments (e.g., conditioned on workloads, drive models, drive age and the total bytes written) to unravel the underlying root causes and actionable advice for practitioners if any.

Second, we pre-screen the raw datasets to avoid bias (e.g., higher average) led by outliers (e.g., overflowing SMART values, NULL `iostat` recordings). Note that on-site engineers have manually verified all failure tickets before this study. In total, we have dropped around 5.8% and 1.5% untrustworthy records from SMART logs and `iostat`, respectively. Moreover, for generalizability concerns, we also exclude drive models with a smaller population (less than 1K) from the study. Note that different suppliers may register a model by different names, but we treat them as the same one here.

Third, we carefully choose statistical instruments to identify and verify the potential patterns in the NVMe SSD failures. Our rationale is that either such techniques or thresholds have been applied in previous studies, or clear documentation indicates the techniques can be used in the targeted scenarios.

3 Baseline Statistics

3.1 Dataset Overview

SMART logs. We begin by presenting the baseline statistics in Table 2, where the dataset is grouped into three categories: Basic Information, Usage Characteristics, and Health Metrics.

Basic Information					Usage Characteristics			Health Metrics				
Model	Cap. (GB)	NAND	Lith./Layer	Total (%)	Drive Years	OP	WAF	Crit. Warn.	CRC Err.	Media Err.	P/E Err.	ARR (%)
I-A	800	MLC	15nm	0.1	3.32	28%	1.69	0.0015 / 0	1439.46 / 0	0 / 0	0 / 0	0.34
	2000	MLC	15nm	0.8	3.07	2%	2.05	0.027 / 0	759.73 / 0	3.52 / 0	0 / 0	0.69
	3840	MLC	15nm	0.1	2.87	7%	0.84	0.0025 / 0	3091.59 / 1	0 / 0	0 / 0	0.78
I-B	1600	MLC	15nm	0.7	2.73	28%	1.82	0.011 / 0	0 / 0	0.01 / 0	0 / 0	1.12
	3200	MLC	15nm	0.1	2.99	28%	1.86	0.16 / 0	0 / 0	759.81 / 0	0 / 0	2.34
I-C	4000	3D-TLC	64L	0.1	0.46	2%	1.04	0 / 0	0 / 0	0 / 0	0 / 0	0.66
II-A	1920	MLC	20nm	0.5	3.44	7%	3.68	0.052 / 0	59.46 / 0	0 / 0	1.70 / 0	0.77
II-B	800	MLC	20nm	0.7	3.60	28%	7.82	0 / 0	52.90 / 0	0 / 0	3.10 / 0	0.49
	1600	MLC	20nm	1.3	3.63	28%	7.97	0 / 0	43.52 / 0	2.69 / 0	5.80 / 0	0.63
II-C	960	3D-TLC	32L	3.4	2.55	7%	3.62	0 / 0	1572.77 / 0	0 / 0	0.79 / 0	0.52
	1920	3D-TLC	32L	1.8	2.50	7%	2.88	0.0017 / 0	849.99 / 0	0.49 / 0	1.60 / 0	0.79
	4000	3D-TLC	32L	5.5	2.39	2%	3.36	0.00079 / 0	957.86 / 0	0.34 / 0	3.60 / 1	0.64
II-D	960	3D-TLC	64L	4.9	1.47	7%	2.45	0.00026 / 0	38.66 / 0	1.45 / 0	0.38 / 0	0.26
	1920	3D-TLC	64L	8.4	0.97	7%	2.37	0.00031 / 0	54.56 / 0	0.45 / 0	0.45 / 0	0.56
	3840	3D-TLC	64L	45.3	0.69	7%	1.96	0.000038 / 0	32.72 / 0	5.53 / 0	0.66 / 0	1.12
II-E	370	NEW	20nm	0.5	1.24	0%	-	0 / 0	72.05 / 0	0.71 / 0	0 / 0	1.40
	750	NEW	20nm	0.7	0.18	0%	-	0 / 0	38.92 / 0	16.27 / 0	0 / 0	3.27
III-A	3200	3D-TLC	48L	0.3	2.65	28%	2.59	0 / 0	19.39 / 0	45.28 / 0	0.28 / 0	2.31
III-B	960	3D-TLC	48L	3.4	1.96	7%	3.34	0.0038 / 0	296.41 / 0	2.29 / 0	30.00 / 0	0.60
	1900	3D-TLC	48L	7.4	1.73	7%	2.78	0.0080 / 0	263.04 / 6	0.82 / 0	69.00 / 0	0.69
	3800	3D-TLC	48L	9.9	1.93	7%	1.87	0.010 / 0	469.66 / 6	1.81 / 0	67.00 / 0	1.13
III-C	960	3D-TLC	64L	4.1	0.45	7%	3.96	0.0023 / 0	124.55 / 0	0.02 / 0	5.30 / 0	0.49

Table 2: Baseline statistics of our drives (§3). The table shows the summarized statistics of NVMe SSD fleet. *Cap.:* capacity; *NAND:* flash architecture; *Lith./Layer:* lithography or numbers of stacking layers; *OP:* Over-Provisioning rate; *WAF:* Write Amplification Factor; *Crit. Warn.:* critical warning; *P/E Err.:* program/erase error. In Health Metrics, the two values separated by a slash refer to mean and median values, respectively.

In Basic Information, we name the drive models as manufacturer-model and use the alphabetic order to refer to the generations of a manufacturer (e.g., I-A stands for the earliest model from manufacturer I). Each model can be further specified by capacities (i.e., Cap. column) and NAND architecture (i.e., NAND column). We mark the planar chips with their lithography (e.g., 15nm for I-A) and the 3D chips with their vertical stacking layers (e.g., 64-layer for I-C). II-E is a unique case as it adopts a novel (neither planar nor 3D stacking) cell, thus named NEW (for anonymity). Finally, we list each model’s relative population (i.e., Total%).

The Usage Characteristics describes the high-level administrative information. The first column is the average power-on time in terms of years. The second and third columns respectively present the over-provisioning rate (i.e., OP) and the calculated average Write Amplification Factor (i.e., WAF). WAF is calculated by dividing the number of NAND writes by the number of logical writes. Both numbers are reported by the SSD SMART attributes.

Last, we cover five primary reliability-related metrics.

- *Critical Warning*, introduced by NVM Express [20], indicates that the drive may have serious media errors (i.e., in read-only or degraded mode), possible hardware failures, or exceeding temperature alarm threshold.

- *CRC Error* refers to the number of transmission errors (e.g., the faulty interconnection between the drive and the host).
- *Media Error* refers to the number of data corruption errors (i.e., unable to access stored data in flash media).
- *Program/Erase Error* refers to the number of flash cell programming errors (e.g., unable to program flash cells from a block that is about to be garbage collected during copyback).
- *Annual Replacement Rate (ARR)* is the number of device failures divided by numbers of device years, reflecting the general reliability of drives (a common standard [34, 41]).

Note that readings of the first four health metrics are heavily biased where zeros account for an absolute majority (e.g., 99.97% for critical warning) of valid recordings. Hence, we list both average and median values (i.e., average/median).

Failure tickets. A subset (around 35%) of our drives details the direct cause upon failure reporting. Here, we present the distribution of their symptoms in Table 3. There are a total of five failure symptoms. I/O failure refers to a drive that fails to perform a read/write request. The Link failure indicates either a connection error during the PCI-e transmission or an abnormal bandwidth. The Lost failure refers to a functioning drive to become unfound. The Boot failure describes a drive that fails to initiate (e.g., mounting file system). The Thres. failure refers to one or more SMART attributes to have reached the

Type	Distribution Statistics				
	Dist.	ARR	ARR_M	ARR_3D	ARR_N
I/O	49.55%	0.40%	0.14%	0.42%	1.07%
Link	11.07%	0.09%	0.01%	0.10%	0.10%
Lost	5.65%	0.05%	0.06%	0.04%	0.01%
Boot	19.59%	0.16%	0.30%	0.14%	0.39%
Thres.	14.15%	0.11%	0.20%	0.10%	0.10%

Table 3: Failure symptom distribution (§3.1). *Dist.*: distribution; *ARR*: overall ARR; *ARR_X*: ARR of drives under different flash architecture; *I/O*: read/write failure; *Link*: connection failure; *Lost*: drive unfound; *Boot*: booting failure; *Thres.*: SMART value over a pre-defined threshold.

pre-defined threshold(s). For each type of failure, we present its distribution (Dist. column) along with the corresponding ARRs in all NVMe SSDs (ARR), MLC-based (ARR_M), 3D-TLC (ARR_3D), and NEW-NAND ones (ARR_N).

3.2 High Level Observations

Based on Tables 2 and 3, we now associate drive characteristics with health metrics to get a high-level understanding of the NVMe SSD fail-stop failures. Note that, even for the same model, the drive population can be a diverse mix of model and usage characteristics (e.g., NAND type, age, and total bytes written). We have further verified our observations through controlled-variable experiments on such impacting factors.

NVMe SSD vs. SATA/SAS SSD. The ARR of NVMe SSD in our dataset is much higher than that of SATA/SAS SSD from Netapp’s enterprise storage systems [34]. We perform a t-test on both sets of ARRs and the corresponding p-value is equal to $3.554e-07$. The average and median ARR of our NVMe SSD are 0.98% and 0.69%, which are $2.77\times$ and $2.83\times$ higher than those of SATA/SAS SSD respectively (i.e., 0.26% and 0.18% calculated from Table 1 in [34]). We obtain similar results as we further break down the SSD population by NAND types and lithography. Regarding Alibaba’s data centers [19], the trend persists except for two models (i.e., C1 and C2 from Table 1 in [19]).

Moreover, we also compare the failure symptom distribution between NVMe SSD and SATA SSD (see Table 3 in [47]). We observe radical changes as I/O error becomes far more prevalent in NVMe SSD (accounting for 49.55%) while Lost error (i.e., drive unfound) is no longer dominant (i.e., 5.65% in NVMe SSD vs. 53.7% in SATA SSD).

Drive capacity. Within the same drive family, the average P/E errors and ARR are positively correlated with the capacity. For example, in the II-D drive family, as the capacity increases, the average P/E error rises from 0.38 to 0.66 and the ARR surges from 0.26% to 1.12%. This is understandable as drives with larger capacity are more likely to be accessed, thereby increasing the chances of suffering program errors.

NAND type. In our dataset, we find that the ARR of 3D-

TLC drives is slightly lower than that of MLC drives, while in SATA SSD [34], the trend is reversed. The ARR of our MLC drives varies from 0.34% to 2.34%, while that of 3D-TLC SSD is from 0.26% to 2.31%. However, we notice that drives with NEW NAND architecture (i.e., II-E family) exhibit around $1.61\times$ and $1.87\times$ higher average ARR than those of MLC and 3D-TLC drives, respectively (the p-values are equal to $2.065e-02$ and $4.351e-03$). Table 3 further demonstrates an ARR breakdown of failure symptoms among different NAND chips. For NEW-based SSD, the main culprits of its high ARR are the I/O and booting failures.

4 The Fail-stop

Now, we present the three major changes in failure patterns in NVMe SSD. For each aspect, we start with existing patterns in SATA/SAS SSD. We then study the differences in NVMe SSD using a similar setup and further verify our findings under a series of controlled-variable experiments.

4.1 Infant Mortality

Finding 1. *Infant mortality, a notorious failure trend in hardware early deployment period, is not notable in NVMe SSD.*

Existing patterns. The Bathtub Curve [40] is a classic depiction of hardware failure variances through time. Generally, there are three main phases: infant mortality, stable (aka. useful-life) and wear-out period. Previous SATA SSD studies suggest that flash drive also follows this trend [35] (i.e., with an early detection phase followed by the bathtub curve).

Difference in NVMe SSD. For NVMe SSD, we are interested in whether such observation still holds. Here, we adopt the monthly failure conditional probability (FCP) to demonstrate the failure trend (i.e., the same metric as in Section 5.1 of [34]). The FCP is calculated as the number of drives to be replaced that month divided by the number of drives surviving that month. In Figure 1, we present a comparison between the six most popular drive families covering varying NAND architectures (i.e., 15-20nm MLC, 32-64 layers 3D-TLC, and the NEW). Throughout the paper, error bars refer to 95% confidence intervals with bootstrap methods [12] (2,000 iterations). In Figure 1, to calculate the error bar of FCP in month x (with N drives surviving month $x-1$), we create 2,000 random samples of FCPs; in each sample, the FCP is calculated based on a randomly-chosen set of N drives (i.e., sampling with replacement). Finally, we calculate the 95% confidence interval based on these 2,000 samples of FCPs as the corresponding error bar.

Should the NVMe SSD still follow the bathtub curve, we shall see the FCP, starting from a high value (i.e., the infant mortality), quickly decreases to a stable range (i.e., the useful life) until surging in the wear-out period. However, from visual inspection, we discover that most drive families do not have outstanding infant mortality during early periods. We further calculate the average FCP under various periods (e.g., 1st to 3rd month and the most recent three months).

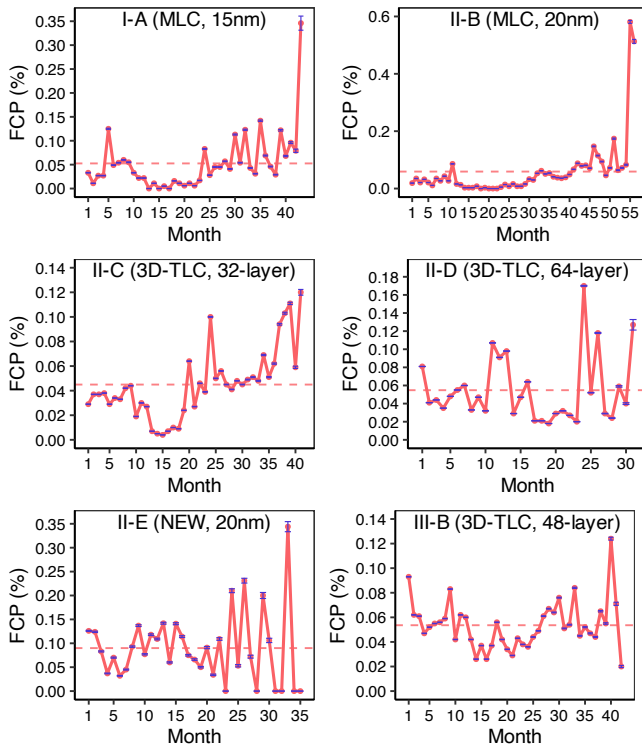


Figure 1: Failure trend (§4.1). The figures present the Failure Conditional Probability (%) vs. drive's deployed time in months in 6 drive families. The dashed line indicates the average value within each figure. Error bars throughout the paper refer to 95% bootstrap confidence intervals [12] (2,000 iterations).

As a result, the early period (i.e., the first three months) has equivalent or even lower FCP than the later periods. For example, in I-A, the first three months yield an average FCP of 0.02%, whereas the average FCPs of the next 9 months (months 4-12) and most recent three months are 0.05% and 0.35%, respectively.

Validity analysis. Next, we explore the reason behind this reliability improvement. First, it is unlikely that the stress tests weed out the faulty drives before deployment. Stress tests are usually short (up to one week) and thus not enough for drives suffering infant mortality (the period lasts for 12-15 months long [35]). Second, we exclude the possibility of external impacts (e.g., uneven distribution of workloads and drive age, if any) using two-sided t-tests. Then, we focus on the internal aspect by studying the SMART attributes variances over time. Here, for both II-D and III-B in Figure 2, nearly all health-related metrics experience the infant mortality as they start with a much higher value and then decrease to a stable range over time. Other drive families, in general, also follow this trend. Note that the values in Figure 2 are normalized to the lowest point on each curve and reported in logarithmic reporting. This indicates that NVMe SSD still

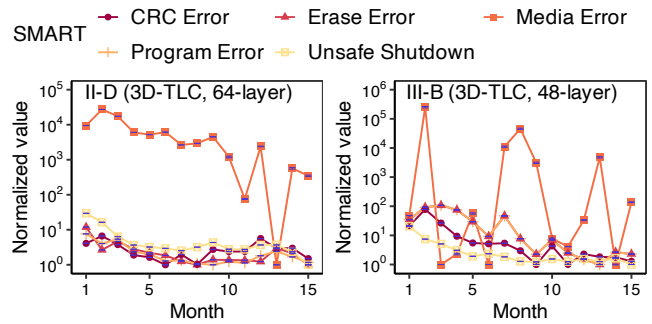


Figure 2: SMART in early age (§4.1). The figures present the average number of SMART-recorded errors per month during the first 15 months of deployment in two major drive models. Note that the numbers are normalized to the lowest point on each curve.

accumulates a large number of errors during the early period. Therefore, we assume it is likely that the improvement of FTL error handling makes the NVMe SSD more resilient in the early periods.

Operational advice. We believe the recent advancement in failure handling has set the NVMe SSDs free from suffering infant mortality. This can serve as a relief signal for the supply chain and the on-site administrators as previous practice usually demands that the cloud operators stockpile extra pieces before initial deployment.

4.2 WAF

Finding 2. NVMe SSD becomes more robust to high write amplification ($WAF > 2$), but extremely low write amplification ($WAF \leq 1$) is still rare-but-deadly.

Existing patterns. Write amplification is a common phenomenon in SSD I/O where the logical writes incur extra data to be written to NAND due to SSD internal operations (e.g., garbage collection and alignment). A higher write amplification factor (i.e., NAND writes size divided by logical writes size) therefore indicates a more random and small-writes-dominant workload. To overcome this disadvantage, manufacturers often use write compression techniques to combine small or buffer repeated writes [9, 46, 49].

Previously, a large-scale SATA SSD failure study by Microsoft pointed out that higher WAF ($WAF > 2$) incurs more SSD failures (i.e., Section 3.5.1 of [36]). Moreover, they suggest that the write compression technique can be damaging where drives with less-than-one WAF have failure rates similar to those with a higher-than-two WAF (see Figure 11 in [36]).

Difference in NVMe SSD. To avoid bias led by model characteristics, we conduct a comparative study within each model family. For each drive family, we first place SSDs into different buckets by WAF with a step of 0.5. Then for each bucket, we calculate its corresponding ARR. Since the 95th percentile of WAF in our entire fleet is around 4, the last bucket includes

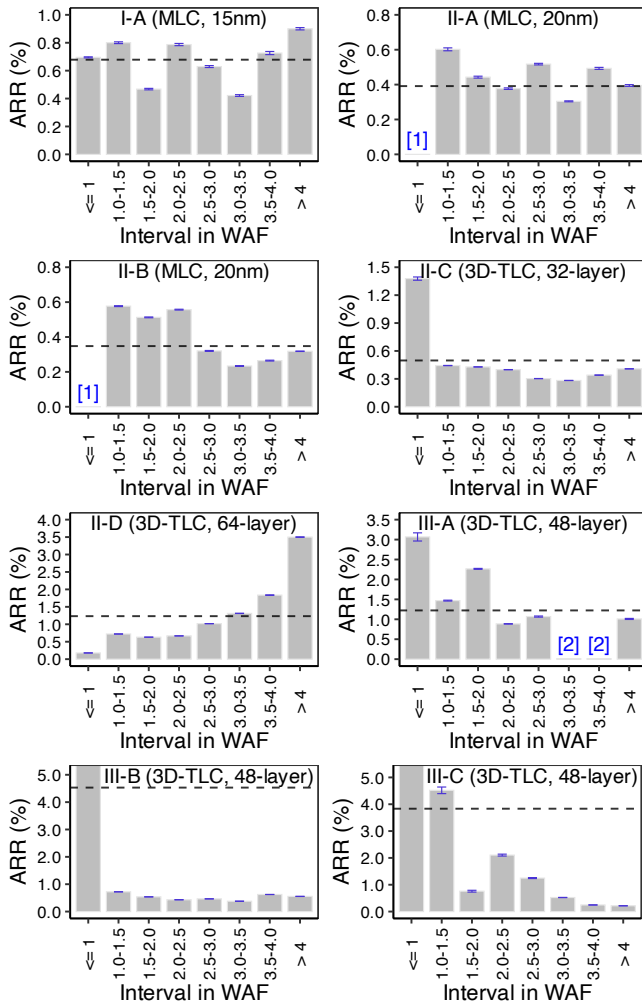


Figure 3: ARR under different WAF levels (§4.2). The grey dashed line indicates the average value within each figure. “[1]”: the bucket includes no drives; “[2]”: the calculated ARR is zero (i.e., no failed drives). The first buckets of III-B and III-C are overflowed ($>5\%$).

drives with WAF above 4. The $WAF \leq 1$ bucket contains drives significantly influenced by the write compression technique.

Figure 3 presents the correlation between WAF and failure rates among eight popular drive families, covering different types of NAND and manufacturers. Here, we make two observations. First, for WAF higher than one, we do not observe a strong positive correlation between WAF and ARR in most drive families (the II-D drive family is considered an exception). A set of Spearman’s Rank Correlation Coefficient tests [44] further statistically confirm our hypothesis. This indicates that NVMe SSD is less affected by random small writes (a major cause for high write amplification). Second, we observe that, for drives with low WAF (i.e., $WAF \leq 1$), their failure rates are still relatively high. On average, these low-WAF drives can have a $2.19\times$ higher ARR rate than average.

Time	Type	SATA SSD	NVMe SSD	Hypo.
Total	node rack	4.5-73.7% 28.6-91.4%	70.6-96.6% 79.4-97.6%	0.04-9.0% 27.8-77.4%
(0, 1min]	node rack	0.8-24.7% 1.7-27.2%	1.1-17.9% 1.3-17.9%	0% 0%
(1d, 1mon]	node rack	1.1-39.4% 6.5-47.9%	14.3-57.5% 15.5-57.2%	0.01-1.1% 2.9-10.0%

Table 4: Intra-node/rack failure distributions across drive types (§4.3). The table presents the relative percentage of intra-node and intra-rack failures in SATA SSD from a previous study [19], our NVMe SSD fleet, and a hypothetical setting where failures are independent of time and location.

An extreme example is the III-B drive family (lower left in Figure 3), where low-WAF drives are $6.18\times$ more likely to fail than average. Fortunately, we discover that these low-WAF drives usually occupy only a small proportion (e.g., only 0.09% in III-B). Even for I-A-3840 (i.e., having an average WAF of 0.84), we argue that low-WAF drives can be easily singled out with simple SMART attributes calculation for close monitoring or reallocation.

Validity analysis. Many factors (e.g., workloads, age, wear, and drive model) can influence both the WAF and reliability. To verify our finding, we further conduct a series of experiments by controlling the above variables (not shown due to space limitations). Our evaluation further confirms that none of the factors influences our finding. Therefore, we conclude that, in NVMe SSD, while the low WAF may still be deadly, the high WAF is no longer concerning.

4.3 Intra-node/rack Failures

Finding 3. Spatially correlated (intra-node/rack) NVMe SSD failures are temporally correlated in the long-term span (i.e., 1 day to 1 month), but no longer prevalent in the short span.

Existing patterns. Correlated drive failures are notorious for their cascading impact on the reliability of the entire distributed system (e.g., reduced redundancy) [2, 15]. According to a previous study in Alibaba (i.e., Finding 5 of [19]), a non-negligible proportion (i.e., up to 34.3% and 44.2%) of SATA SSD spatial-correlated (intra-node/rack) failures can also be temporal-correlated within a short span (i.e., at most one minute apart), posing a critical challenge to the overall stability.

Difference in NVMe SSD. To study whether such a correlated pattern still plagues the NVMe SSD fleet, we further check the intra-node/rack failure time intervals in our datasets. Here, to be consistent with the previous study [19], we reuse the Relative Percentage of Failures (RPF) to calculate the likelihood of correlated failures. In RPF, the numerator is the number of the sets of failures that occur between a specific period (e.g., 0 to 1 minute). The denominator is the sum of all failures of a particular drive model. Note that, in RPF, the same failure can be counted repeatedly as a member of

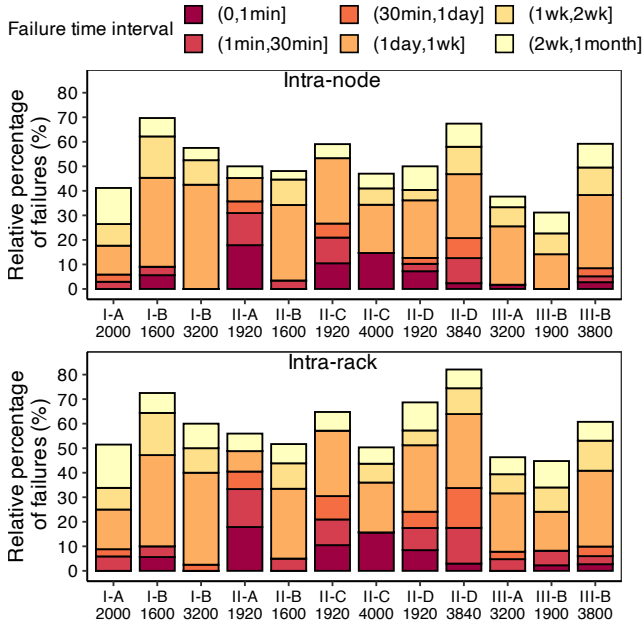


Figure 4: Intra-node/rack failure distributions across drive models (§4.3). The figures present relative percentage of intra-node and intra-rack failures for each drive model. Note that stacked bars can reach above 100% because failures could be counted multiple times into different buckets.

different correlated failures sets. For example, consider three failures, say A, B, and C, all occur within one minute inside the same node. Then there are three sets of correlated failures (i.e., [A,B], [B,C], and [A,C]), thus yielding an RPF of 100%.

In Table 4, we list the RPF from the previous study (i.e., SATA SSD column) and ours (i.e., NVMe SSD column). We make the following observations. First, the accumulated RPFs (i.e., Total row) across all NVMe drive models are substantially higher, with an increase up to $14.69\times$ and $1.78\times$ for intra-node and intra-rack scenarios, respectively. The corresponding t-tests return p-values equal to $6.322e-06$ and $1.881e-04$.

Second, unlike SATA SSD (mostly correlated during short intervals), correlated failures in NVMe SSD are commonly observed only in long intervals (i.e., 1 day to 1 month). Here, we use Figure 4 to demonstrate further the distribution of correlated failure intervals among different models on a weekly breakdown of the long intervals (i.e., 1 day to 1 month). In Figure 4, the upper graph shows the distribution of intra-node intervals, and the lower graph shows the intra-rack ones. We denote each interval with a different color (e.g., darkest for the shortest interval). Figure 4 shows the prevalence of long intervals (i.e., the three lightest/top boxes in each bar) in correlated failures. Conversely, short and medium ones (i.e., the three darkest/bottom boxes) on average occupy less than 17.86% (within 1 minute), 15.48% (1-30 min), and 16.25% (30 min to 1 day) of the total RPF.

Validity analysis. Each rack in our setup hosts hundreds of drives. Under such a considerable number, uniformly distributed failures may also co-occur within a rack. To verify that intra-node/rack failures are indeed a result of non-uniformity, we perform a set of hypothetical experiments where drive failures are redistributed to be uncorrelated (i.e., independent of location and arrival time). First, for each drive model, we sample without replacement to get a new batch of drives and mark them as “failed” drives. Second, we assign a random timestamp from 2019-11-04 to 2020-11-02 to each “failed” drive as its failure time (see Table 1). For a fair analysis, we repeat the above procedures 2,000 times and calculate the average RPFs of intra-node/rack failures for each drive model.

In Table 4, we compare the intra-node/rack failure distributions of hypothetical experiments (i.e., Hypo. column) with those of the original setting (i.e., NVMe SSD column). Our observations are as follows. First, intra-node failures under the hypothetical setting are nearly negligible. For example, the accumulated RPFs (i.e., Total row) of intra-node failures are only 0.04-9.0% under the hypothetical setting, whereas those from the original setting (70.6-96.6%) are much smaller (the p-value is less than $2.2e-16$). Second, even though intra-rack failures are non-negligible under the hypothetical setting, their RPFs are consistently smaller than those from the original setting, e.g., with accumulated RPFs of 27.8-77.4% vs. 79.4-97.6% (the p-value is equal to $1.119e-08$). Therefore, the non-uniformity in intra-node/rack failures is significant in our dataset in contrast with the hypothetical setting.

Operational advice. While the decline of closely correlated failures implies a lower risk of experiencing system-wide failures, the surging of long-interval correlated failures still poses a pressing threat. An inconvenient fact is that fixing drive failure usually starts with software-based approaches (e.g., data scrubbing and fsck), and such online checking and repairing takes time [16, 33, 47]. In fact, we discover that 43.90%, 14.36%, and 10.90% of the failed drives in our clusters are repaired after one day, one week, and two weeks. Based on our finding, we have refined our operational process by directly putting drives offline upon failures to reduce the chances of suffering long-term correlated failures.

5 The Fail-slow

Apart from the common fail-stop failures, we are also interested in fail-slow failures where drives exhibit performance much less than expected (e.g., considerably high latency under normal traffic). We hypothesize that the ultra-low latency nature of NVMe SSD would make the drive more *susceptible* to fail-slow failures. To verify our assumption, we conducted an extensive study based on the per drive `iostat` traces from more than half a million NVMe SSDs and more than 4 million HDDs during four-month monitoring.

Note that our storage system requires all replicas (three replicas in most cases) ACKed before any write request is

Model	Lith./Layer/Type	Slow Drive (%)	Event Freq.	Dur. (min)	Event Laten. (us)	Slow-down Ratio	Slow Drive (%)	Event Freq.	Dur. (min)	Event Laten. (us)	Slow-down Ratio
						5min					
I-A-2000	15nm	4.44%	225.06	18.98	195.60	2.39	3.65%	118.01	38.44	200.20	2.39
II-A-1920	20nm	1.25%	24.22	8.60	152.77	1.94	0.57%	8.70	20.36	148.96	1.80
II-C-1920	32L	0.52%	23.50	19.31	263.31	2.19	0.37%	11.84	39.67	256.58	2.19
II-C-4000	32L	0.06%	1.59	8.41	180.67	2.04	0.05%	0.66	21.16	175.62	2.03
II-D-1920	64L	0.17%	4.52	22.00	34.98	2.30	0.12%	2.30	42.59	36.22	2.35
II-D-3840	64L	0.48%	14.08	12.00	152.63	5.87	0.31%	5.99	27.08	122.35	4.48
III-B-1900	48L	3.04%	46.75	9.43	54.67	2.19	1.55%	12.82	24.69	56.68	2.22
III-B-3800	48L	1.31%	44.28	13.51	360.59	6.33	1.05%	20.89	30.39	244.74	4.29
Average	-	1.41%	48.00	14.03	174.40	3.16	0.96%	22.65	30.55	155.17	2.72
H1	CMR	0.32%	4.53	8.56	47370.67	2.18	0.09%	1.15	23.97	55120.74	2.33
H2	CMR	0.24%	2.30	6.92	12355.12	2.04	0.03%	0.32	21.20	14796.03	2.38
H3	CMR	0.04%	0.51	11.72	1962.49	3.01	0.03%	0.36	28.49	2041.27	3.39
Average	-	0.20%	2.45	9.07	20562.76	2.41	0.05%	0.61	24.55	23986.01	2.70
						30min					
I-A-2000	15nm	3.19%	70.46	60.50	207.18	2.38	2.49%	32.95	97.33	203.25	2.24
II-A-1920	20nm	0.45%	3.03	34.53	143.81	1.81	0.11%	0.38	60.25	147.45	1.79
II-C-1920	32L	0.36%	7.63	60.82	263.97	2.18	0.32%	3.61	100.04	272.49	2.14
II-C-4000	32L	0.03%	0.28	39.85	176.39	2.03	0.01%	0.03	97.88	182.67	2.20
II-D-1920	64L	0.08%	1.26	61.72	37.57	2.40	0.04%	0.52	92.95	39.62	2.49
II-D-3840	64L	0.23%	2.60	47.63	132.95	4.31	0.13%	0.85	86.96	128.33	3.62
III-B-1900	48L	0.75%	4.74	48.27	61.94	2.35	0.40%	1.96	86.63	74.55	2.70
III-B-3800	48L	0.91%	11.18	49.53	248.33	4.18	0.63%	4.14	84.23	148.26	2.32
Average	-	0.75%	12.65	50.36	159.02	2.71	0.52%	5.56	88.28	149.58	2.44
H1	CMR	0.04%	0.41	44.79	58673.31	2.43	0.02%	0.11	83.36	62272.65	2.50
H2	CMR	0.01%	0.10	39.86	15496.51	2.55	<0.01%	0.02	98.31	20991.78	3.66
H3	CMR	0.03%	0.21	53.05	2937.19	3.66	0.02%	0.11	96.27	3187.88	5.09
Average	-	0.03%	0.24	45.90	25702.34	2.88	0.01%	0.08	92.65	28817.44	3.75
						60min					
I-A-2000	15nm	3.19%	70.46	60.50	207.18	2.38	2.49%	32.95	97.33	203.25	2.24
II-A-1920	20nm	0.45%	3.03	34.53	143.81	1.81	0.11%	0.38	60.25	147.45	1.79
II-C-1920	32L	0.36%	7.63	60.82	263.97	2.18	0.32%	3.61	100.04	272.49	2.14
II-C-4000	32L	0.03%	0.28	39.85	176.39	2.03	0.01%	0.03	97.88	182.67	2.20
II-D-1920	64L	0.08%	1.26	61.72	37.57	2.40	0.04%	0.52	92.95	39.62	2.49
II-D-3840	64L	0.23%	2.60	47.63	132.95	4.31	0.13%	0.85	86.96	128.33	3.62
III-B-1900	48L	0.75%	4.74	48.27	61.94	2.35	0.40%	1.96	86.63	74.55	2.70
III-B-3800	48L	0.91%	11.18	49.53	248.33	4.18	0.63%	4.14	84.23	148.26	2.32
Average	-	0.75%	12.65	50.36	159.02	2.71	0.52%	5.56	88.28	149.58	2.44
H1	CMR	0.04%	0.41	44.79	58673.31	2.43	0.02%	0.11	83.36	62272.65	2.50
H2	CMR	0.01%	0.10	39.86	15496.51	2.55	<0.01%	0.02	98.31	20991.78	3.66
H3	CMR	0.03%	0.21	53.05	2937.19	3.66	0.02%	0.11	96.27	3187.88	5.09
Average	-	0.03%	0.24	45.90	25702.34	2.88	0.01%	0.08	92.65	28817.44	3.75

Table 5: Baseline statistics for fail-slow (§5.1-§5.2). The table shows the summarized statistics of fail-slow occurrences under the 5-min to 60-min duration requirements. **Lith./Layer/Type:** lithography, numbers of stacking layers or HDD type; **Event Freq.:** fail-slow event frequency per 1K drives per hour; **Dur.:** average event duration in minutes; **Event Laten.:** average event latency in μs . Note that the SSDs and HDDs under heavy traffic are not included (see §5.1).

returned, while only one ACKed for each read request. Thus, fail-slow failures are more likely to impact the write performance than the read. Such a phenomenon agrees with most known fail-slow cases in practice. Throughout this section, we focus on the write latency where fail-slow failure can be more destructive.

5.1 Identifying Fail-slow Events and Drives

Currently, a subset of our clusters is equipped with daemons to monitor the `iostat` of the deployed drives. Due to capacity limits and performance concerns, the daemon runs 3 hours (9 P.M. to 12 A.M.) each day. It only records the average statistics of each monitoring window (15 seconds in the current setup), thereby yielding 720 records per drive (3 hours divided by 15 seconds) each day.

Methodology overview. We use the following threshold-based approach to identify fail-slow drives (similar to a previous study on SATA SSD and HDD tail latency [21]). The first step is to select suspicious drives with high latencies. Then,

we determine whether the chosen drives are indeed fail-slow by checking the existence of consistent slowdowns.

Identifying suspicious fail-slow drives. In the first step, we observe that the performance (e.g., latency, IOPS, and throughput) records within a cluster generally follow a Positively Skewed Distribution. For example, in one cluster, the median latency is only $49.19\mu s$ while the average latency is $667.85\mu s$. Thus, we can use a latency threshold as $(-\infty, 3rd_quartile + 2IQR)$ to identify the outliers (i.e., slow drives) [39] where the *IQR* (interquartile range) is computed by subtracting the first quartile from the third quartile.

If the 3-hour median latency of a drive is beyond the bar, we mark this drive as a suspicious slow drive. To avoid reporting led by heavy traffic (i.e., false-positive), we also rule out high-latency drives under heavy traffic (i.e., IOPS/throughput is also beyond $3rd_quartile + 2IQR$).

Identifying slowdown events. Note that a suspicious drive may be marked due to transient but time-consuming events (e.g., read retries, unstable connection). Therefore, we further

check whether a suspicious drive has experienced a consistent slowdown event to pinpoint the fail-slow drives. Here, we borrow the idea of measuring slowdown events from a previous SSD performance study [21].

First, we mark the 3-hour `iostat` latency records from the suspicious slow drive and its 11 intra-node peers (12 drives per node) as L_i^k , representing the record i ($i \in \{1, 2, \dots, 720\}$) from drive k ($k \in \{1, 2, \dots, 12\}$). Then, we use RL_i^k (Relative Latency) to indicate the slowdown degree of drive k at record i , formally $RL_i^k = \frac{L_i^k}{\text{median}(L_i^1, L_i^2, \dots, L_i^{12})}$. Then, we formulate an event as $E_{i,j}^k$ by computing the means of RLs of drive k from record i to j , formally $E_{i,j}^k = \text{mean}(RL_i^k, RL_{i+1}^k, \dots, RL_j^k)$.

For an $E_{i,j}^k$ to be considered a fail-slow event, it must satisfy two requirements. First, $E_{i,j}^k$ needs to be larger than an empirical slowdown degree. We set the slowdown degree as 2 (same in [21]), meaning that, during the event, the victim drive is at least twice slower than its peers. Second, we set four minimum spans as 5, 15, 30 and 60 minutes, meaning the $E_{i,j}^k$ should last longer than 20, 60, 120 or 240 records (a record spans 15 seconds).

To sum up, a drive (i.e., NVMe SSD or HDD) is deemed fail-slow if and only if it has a high median latency (i.e., higher than top 0.04% latency variances in the cluster during 3-hour monitoring) and suffers at least one fail-slow event.

5.2 Dataset and High Level Observations

5.2.1 Dataset Overview

In total, we have identified around 5K and 3K fail-slow NVMe SSDs and HDDs, respectively. Table 5 gives an overview of the fail-slow drives and events among the fleets. We use four quadrants to represent the statistics under different requirements (i.e., 5 to 60 min). The upper half of each quadrant includes 8 major NVMe SSD models (named as brand-model-capacity), while the lower includes the three most popular HDD models (H1, H2, and H3) in our clusters.

For each column, we begin by listing the lithography (for planar NAND), layers (for 3D-NAND), or the type (for HDD, i.e., CMR or SMR). Further, we show the percentage of drives that have been identified as fail-slow ones in that model (Slow Drive%). The Event Freq. describes the numbers of events per 1000 drives per hour, reflecting the fail-slow severity in a mid-sized cluster. The following two columns (Duration and Event Latency) show the average fail-slow event duration and average event latency. The final column (Slowdown Ratio) is the ratio of average event latency to average latency of peer drives (i.e., healthy drives from the same node) during the event. The last row of each sub-quadrant is the average value for each category (i.e., SSD or HDD).

5.2.2 SSD vs. HDD

Finding 4. *Compared to HDD, fail-slow failure in NVMe SSD is much more widespread and frequent, and can degrade the drive to SATA SSD or even HDD level performance.*

We start with the differences between NVMe SSDs and HDDs. First, we observe the disparity between HDD and SSD in slow drive popularity (i.e., Slow Drive %). Comparing the average row in each quadrant reveals that slow drives are $6.05\times$ (i.e., 1.41% to 0.20% in the 5-min quadrant) to $51\times$ (0.52% to 0.01% in the 60-min quadrant) more common in SSDs. Similarly, we also observe that the fail-slow occurrences (i.e., Event Freq.) are much more frequent in SSDs, ranging from $18.59\times$ (5-min) to $68.50\times$ (60-min).

Regarding the event duration, the difference varies. On average, the SSD event lasts up to 55% longer in the first three quadrants, but the trend reverses as the HDD event costs 5% more time in the 60-min quadrant. This indicates that fail-slow events in NVMe SSD are relatively short-termed.

Moreover, while models like II-D-1920 and III-B-1900 still deliver relatively satisfying performance, fail-slow NVMe SSDs usually degrade to SATA-SSD-level latency (i.e., having an average event latency around $160\mu\text{s}$, see average rows of event latency from 5-min to 60-min). Even worse, our evaluation shows that the top 1% slowest events in several NVMe SSD models deteriorate to an average latency of around 22ms, an unsatisfying performance even for HDD.

The comparisons of slow drive popularity, event frequency, and performance prove that the NVMe SSD is indeed widely plagued (1.41% affected under 5-min requirement) and severely impacted ($\sim 160\mu\text{s}$ average event latency) by fail-slow failure. Recall that the dataset comes from only three hours of monitoring per day for four months. Plus, all of our models are enterprise-level, and we have already excluded SSDs under heavy traffic. Therefore, we expect the annual fail-slow drive rate to be higher and fail-slow occurrences more frequent in the field.

Operational advice. Experiencing widespread and severe fail-slow faults can be particularly harmful to NVMe SSDs as performance-sensitive jobs are usually placed on them. However, simply putting all fail-slow drives offline can be unacceptably expensive. Recently, we have been experimenting with a “three strikes” approach to tackle the suspiciously slow drives. Specifically, the first time a drive is diagnosed with fail-slow failure, we would clean the drive’s data and deploy it again as a new drive. In the second time, we would fully flush the drive with zeroes, reformat and redeploy it. A third timer would be directly put offline for replacement. Unfortunately, we have just deployed this strategy and do not have enough samples for analysis.

Root cause. We have sent 100 slowest SSDs (around the top 2% of the identified slow drives with an average event latency of 4.4 ms) back to vendors for repair. The results show that 33 of them have bad capacitors, causing the malfunctioning buffer and thus the high latency. 46 of them contain bad chips and the root causes of the rest remain unclear.

5.2.3 Differences between SSD models.

Finding 5. *The manufacturer is a dominant factor of fail-slow*

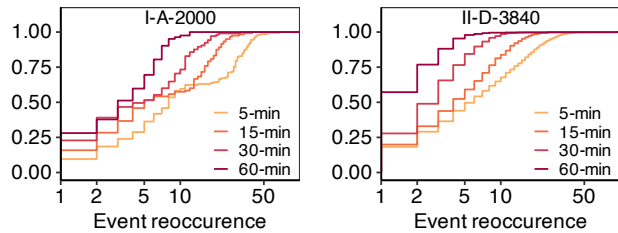


Figure 5: CDF of fail-slow event reoccurrence (§5.2). The figures present the distribution of event reoccurrence for two models under four duration requirements.

drive population in NVMe SSD.

Now, we dig deeper by focusing on the fail-slow distribution differences between SSD models. First, we look at the influences of the manufacturers. Our dataset includes three manufacturers (i.e., I, II, and III). For slow drive percentage, there is a clear order (i.e., manufacturer I followed by III and II) across the four quadrants. Even the highest value of III (e.g., 3.04% of III-B-1900 in 5-min quadrant) is well behind that of the I’s model (i.e., 4.44% of I-A-2000), which also applies to the comparison between III and II. However, we do not observe visible patterns for the event duration, event latency, and slowdown ratio.

Finding 6. Higher fail-slow drive popularity does not always lead to a higher fail-slow event frequency.

Moreover, we notice a seemingly counter-intuitive pattern. One may assume a higher fail-slow drive percentage leads to a higher event frequency. While this hypothesis holds in the longest duration requirement (60-min quadrant), we find many counter-examples among shorter ones (e.g., II-A-1920 and II-C-1920 in 5-min quadrant).

A possible explanation is that under a shorter duration requirement, there are more drives with multiple events, resulting in a small slow drive percentage with high event frequency. Here, we further verify this assumption by using Figure 5, a CDF of events per drive under different duration requirements. We can clearly see that drives under shorter durations accumulate more events than those under longer durations.

5.3 Correlating Factors

In this section, we conduct an extensive study on fail-slow failures versus various correlating factors (i.e., drive age, workload and SMART attributes).

5.3.1 Drive Age

Finding 7. Fail-slow drive population and event frequency are strongly correlated with age, but only for old (power-on time > 41 months) NVMe SSDs.

Age is widely known for its significant impact on SSD fail-stop failures [34, 35, 41]. We correlate fail-slow metrics with drive power-on time to see the significance of age here.

We first place all fail-slow drives into monthly buckets

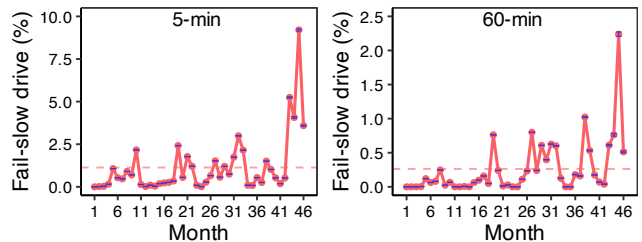


Figure 6: Fail-slow drive percentage across time (§5.3.1). The figures show the percentage of fail-slow drives per month under the 5-min and 60-min duration requirements. The dashed line indicates the average value within each figure.

(e.g., bucket-1 includes fail-slow drives with power-on time between 0 to 1 month). Note that for a drive with multiple event occurrences (e.g., fail-slow events in both 34th and 35th months), we put the drive to the earliest bucket (i.e., 34th-month bucket). Then we calculate the fail-slow drive percentage for each bucket by dividing the numbers of fail-slow drives against the numbers of drives of the same age.

Figure 6 demonstrates the population variances along time under the 5-min (left) and the 60-min (right) requirements where the horizontal dashed line is the average. We can see that the population, in both scenarios, oscillates around the average value at first and then start to surge in the final months. Further, a Spearman’s Rank Correlation Coefficient (SRCC) test [44] reveals that the fail-slow population in old drives (>41 months) is highly correlated with age. Specifically, in the 5-min requirement, the SRCC score for old drives is around 0.92 (way beyond the common threshold for positive correlation, i.e., 0.5). In contrast, the scores from the rest (i.e., younger drives) are close to 0, meaning no correlation. Similar trend exists under 15-min, 30-min, and 60-min requirements.

Next, we adopt similar approaches to measure the correlation between age and other metrics, including event frequency, event duration, and slowdown ratio. We find that the event frequency is similar to the fail-slow population where old drives (i.e., > 41 months) are strongly correlated with age while young drives are not. We do not observe a notable correlation for duration and slowdown ratio, indicating that both metrics remain rather stable throughout the lifecycle.

5.3.2 Workload

Finding 8. The workload can significantly affect various fail-slow characteristics, and heavy traffic workload may have long-lasting impacts on fail-slow occurrences.

Workload is also a well-known impact factor on the SATA SSD fail-stop failures [1, 24, 36, 41, 47]. The key difference between workloads is the I/O pattern. Therefore, we evaluate the impacts of workloads by studying four representative cloud storage services with drastically different access patterns, namely block storage, buffering, object storage, and query.

G	Age-Wr	Wl.	Slow Drive (%)	Event Freq.	Dur. (min)	Slow-down Ratio
II-D-3840						
1	3rd-2nd	Block	0.02	0.23	9.81	1.99
		Buffer	39.17	1318.51	11.85	2.28
		Query	0.08	2.31	6.83	3.01
2	3rd-3rd	Block	0.01	1.84	19.60	2.15
		Buffer	13.86	466.00	13.38	2.22
III-B-3800						
3	2nd-1st	Block	0.03	0.65	15.29	152.59
		Object	5.86	1187.69	26.67	12.41
4	2nd-2nd	Block	0.01	0.15	7.04	2.06
		Buffer	36.88	1196.75	12.00	2.30
5	3rd-2nd	Block	0.71	12.76	10.09	64.91
		Buffer	10.18	608.78	20.24	2.39

Table 6: Fail-slow statistics for groups of workloads (§5.3.2). The table presents fail-slow metrics under different workloads with control variables on the drive model, age, and P/E cycle (total bytes written divided by capacity) under 5-min duration requirement. **G**: variable-controlling group; **Age-Wr**: age-write bucket; **Wl.**: workload; “Slow Drive (%)” to “Slow-down Ratio” follow the same metrics in Table 5.

As factors like age and manufacturers could severely influence the fail-slow failures, we thus conduct this study in a finer granularity by setting multiple variable-controlling groups. In Table 6, based on the fail-slow metrics under 5-min duration requirement, we group the drives (G column) by drive model, age, and P/E cycle (total bytes written divided by capacity). We choose two drive models, II-D-3840 (upper half) and III-B-3800 (lower half), as they are from different manufacturers and both popular among different services. Then, we control other variables as Age-Wr (age and P/E cycle). The age is listed by years and the P/E cycle is broken down into 4 intervals (i.e., ≤ 100 , $100\sim 500$, $500\sim 1K$, and $1K\sim 10K$ P/E cycles, respectively). For example, the 3rd-2nd from group 1 means the drives from this group have a deployment time between 2 to 3 years and a usage level between 100 to 500 P/E cycles. For each group, we further list the workload (Wl. column) and the corresponding fail-slow metrics (“Slow Drive (%)” to “Slow-down Ratio” column, same as Table 5).

Here, we make the following observations. First, by comparing the metrics within each group, we can see that the workload can significantly affect all four fail-slow metrics. For instance, in groups 1 and 2, the fail-slow population and event frequency of buffering workload can be thousands of times higher than those in block storage (e.g., 39.17% vs. 0.02% in group 1). Similar disparities in event duration and slowdown ratio can be observed between block and object storage in group 3, or between block storage and buffering in group 5.

Second, the patterns can preserve despite model, age, or

P/E cycle variances. For example, by comparing groups 2 and 5, while the groups are of different models and usage levels, the buffering workload in both groups has a much higher slow drive percentage and event frequency.

To sum up, the above experiments verify the significant influences of workloads on fail-slow failure. Primarily, we discover that fail-slow failure favors the buffering workload the most. In practice, the drives under the buffering workload usually have constantly heavy traffic (e.g., storing intermediate results of big data workload). Recall that we have already excluded SSDs under heavy traffic from consideration. Therefore, a possible explanation is that the heavy traffic may have a long-lasting effect (e.g., leaving data more scattered), making the drive more susceptible to fail-slow failure.

5.3.3 SMART Attributes

Finding 9. SMART attributes only exhibit negligible correlation with fail-slow metrics.

Now, we analyze whether SMART attributes (an essential set of indicators for fail-stop failures) correlate with fail-slow failures. We collect SMART data on the last day of our four-month fail-slow detection period. Further, we divide drives into groups based on drive model, age, P/E cycle, and workload. Within each group, we label drives as either “slow” or “not-slow”. Finally, we apply SRCC [44] to examine the correlation between fail-slow failures and SMART attributes.

Under the 5-min duration requirement, we obtain 40 groups of drives. None of them exhibit a clear correlation with any SMART attributes, such as Critical Warning, P/E Error, and CRC Error. The above results preserve under 15-min, 30-min and 60-min duration requirements. Even if we set drives with multiple fail-slow events as “slow”, the results remain. Hence, we conclude that the root causes and/or the symptoms of fail-slow failures are not (well) captured by the SMART.

Operational advice. In this case, we decide not to integrate SMART attributes to improve the fail-slow detection. Currently, we have been exploring various approaches. The major hurdle is the lack of verified positive samples (i.e., fail-slow drives) due to the lack of a fail-slow oracle. Therefore, based on the performance, we tried classic statistical methods and discovered that basic linear or polynomial regression is not very practical as they require constant adjustment for the varying traffic (even within the same workload). We leave employing machine learning models as a part of our future work once the “three strikes” yields convincing and abundant cases. Also, we encourage manufacturers to reveal drive characteristics (e.g., flash GC timing) to facilitate fail-slow identification.

5.4 Transition to Failures

Finding 10. The transition from fail-slow to fail-stop failures is rarely observed, at least not observed within a short time interval (within 5 months).

Previous case studies indicate that a fail-slow failure may

	Not-replaced	Replaced	Total
Not-slow	98.84% (770965)	0.57% (4429)	99.41% (775394)
Slow	0.59% (4574)	<0.01% (10)	0.59% (4584)
Total	99.43% (775539)	0.57% (4439)	100% (779978)

Table 7: Transition from fail-slow to fail-stop failures (§5.4). The table presents a contingency table of fail-slow and failed (later in time) drives for NVMe SSDs under the 5-min duration requirement.

turn into a fail-stop failure [17]. Therefore, we collect up-to-date failure tickets ever since the beginning date of our detection period. The latest failure tickets are about 5 months older than the last recorded fail-slow event.

Table 7 is a sample contingency table recording the frequency counts of drives based on 2 categories: appearing in the failure tickets (Replaced column) or not (Not-replaced column) and having at least one fail-slow event (Slow row) or not (Not-slow row). The result is rather surprising as only 10 drives exhibit fail-slow failures before fail-stop failures, yielding a relatively small population in both slow (around 0.22%) and replaced (around 0.23%) drives. The mean and median transition time are 73 and 67 days, respectively. A possible reason is that fail-slow seldom or may need a long time to transit to a fail-stop failure. Therefore, we conclude the fail-slow failures are unlikely to transit to fail-stop failures, at least not within a few months.

6 Limitation

Environmental differences. The main methodology of this paper is to draw side-by-side comparisons with previous findings. The environmental differences (e.g., workload and hardware setup) may impact the validity of our findings. Therefore, throughout the study, we use controlled-variable experiments to rule out the biases led by such influences and only make findings when statistical confidence is enough. Plus, many previous studies share great similarities with our setups (i.e., cloud storage systems from [19, 35, 36, 47]) and workloads (e.g., object storage in [19, 36, 47] and database storage in [36, 47]). Hence, our findings are a result of the NVMe SSD characteristics instead of the environmental factors.

Comprehensiveness. Previous studies have covered various aspects of SATA SSD failures. Due to the space limit, we are unable to present all of them. In the paper, we do not discuss the topics due to three reasons: missing data sources (e.g., bus power consumption [35]), statistically unconvincing results (e.g., lithography [34]), and unchanged failure patterns.

Fail-slow detection. Unlike fail-stop failures where the oracle is clear (i.e., the five symptoms in Table 3), fail-slow failures are often difficult to pinpoint and thus rely on empirical thresholds. In the study, we place a rather strict threshold

and drives under heavy traffic are not considered. The average event latency (close to SATA SSD performance), shown in Table 5, confirms the effectiveness of our detection approach. Even though we may underestimate the impacts of fail-slow occurrences due to the demanding standards, we believe our dataset and findings are sufficient to reveal a rather concerning status quo of fail-slow NVMe SSD in the field.

7 Related Work

SATA/SAS SSD failure study. There are several field studies of SSD failures in large datacenters, including NetApp [34], Google [4, 41], Alibaba [19, 47], Facebook [35], and Microsoft [36]. These studies share important insights regarding the trend, impacting factors, and correlation of the SATA/SAS SSD failures in the field. Our study distinguishes from them in two aspects. First, we focus on NVMe SSD, which can have distinctive failure characteristics due to internal (e.g., RAIN) and external (e.g., NVMe interface) changes. Second, apart from fail-stop failures, we also study the fail-slow failures, a pressing issue especially for the NVMe SSD.

Fail-slow failure study. Fail-slow failure (aka. gray failure) has attracted increasing attention from academia and industry [13, 17, 21, 22, 25, 26, 37]. Specifically, Gunawi et al. [17] collect more than 100 hardware fail-slow cases from various datacenters and perform qualitative analysis to understand the distribution and root causes behind the failures. Moreover, Hao et al. [21] reveal the distribution of tail latency in large-scale SSD/HDD-based RAID systems. Our work is different from the above as we focus on NVMe SSDs inside the general-purpose cloud storage system and perform large-scale quantitative analysis based on the monitoring data.

8 Conclusion

We perform a large-scale failure study of NVMe SSDs in the field. We have identified major changes of NVMe SSD fail-stop failure patterns including failures, robustness under WAF and the temporal correlation. Also, we investigate the fail-slow failures and the impact factors at scale. Altogether, we obtain 10 findings and open-source our dataset.

Acknowledgements

We would like to thank our shepherd and the anonymous reviewers for their insightful comments and suggestions. This research was supported by NSFC (62102424, 61872376, U1736207, 62072306), the Alibaba Innovation Research (AIR) program, National Key R&D Program of China (2018YFB2101102), Program of Hunan Postdoc Innovation (2021RC2069), and Program of Shanghai Academic Research Leader (20XD1402100). The authors thank Ryan Huang, Ennan Zhai, Shiming Wang, and Amber Bi for their feedbacks on early versions of this paper.

References

- [1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX ATC)*, 2008.
- [2] Ramnathan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
- [3] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An Analysis of Network-Partitioning Failures in Cloud Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
- [4] Jacob Alter, Ji Xue, Alma Dimnaku, and Evgenia Smirni. SSD Failures in the Field: Symptoms, Causes, and Prediction Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2019.
- [5] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.
- [6] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2007.
- [7] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawa, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Analyzing the Effects of Disk-pointer Corruption. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2008.
- [8] Matias Björling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC)*, 2021.
- [9] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.
- [10] Brian Choi, Randal Burns, and Peng Huang. Understanding and Dealing with Hard Faults in Persistent Memory Systems. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys)*, 2021.
- [11] Kingston Technology Corporation. SMART Attribute Details. https://media.kingston.com/support/downloads/MKP_306_SMART_attribute.pdf, 2015.
- [12] Thomas DiCiccio and Bradley Efron. Bootstrap confidence intervals. *Statistical science*, 1996.
- [13] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*, 2013.
- [14] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.
- [15] Aishwarya Ganesan, Ramnathan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.
- [16] Om Rameshwar Gatla, Muhammad Hameed, Mai Zheng, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojević, Cyril Guyot, and Robert Mateescu. Towards Robust File System Checkers. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.
- [17] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Gollhofer, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Biralı Runesha, Mingzhe Hao, and Huaicheng Li. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.
- [18] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.
- [19] Shujie Han, Patrick P. C. Lee, Fan Xu, Yi Liu, Cheng He, and Jiongzhou Liu. An In-Depth Study of Correlated

- Failures in Production SSD-Based Data Centers. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, 2021.
- [20] Jonmichael Hands. How SSDs Fail – NVMe™ SSD Management, Error Reporting, and Logging Capabilities. <https://nvmexpress.org/how-ssds-fail-nvme-ssd-management-error-reporting-and-logging-capabilities/>, 2020.
- [21] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.
- [22] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
- [23] Bryan Harris and Nihat Altiparmak. Ultra-Low Latency SSDs’ Impact on Overall Energy Efficiency. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2020.
- [24] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, 2017.
- [25] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and Enhancing In Situ System Observability for Failure Detection. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
- [26] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray Failure: The Achilles’ Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.
- [27] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. Exploring System Challenges of Ultra-Low Latency Solid State Drives. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.
- [28] Gyun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC)*, 2019.
- [29] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Write Dependency Disentanglement with HORAE. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
- [30] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A Multicore-Accelerated File System for Flash Storage. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC)*, 2021.
- [31] Xiaojian Liao, Youyou Lu, Zhe Yang, and Jiwu Shu. Crash Consistent Non-Volatile Memory Express. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.
- [32] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. HeatWatch: Improving 3D NAND Flash Memory Device Reliability by Exploiting Self-Recovery and Temperature Awareness. In *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.
- [33] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ffsck: The Fast File System Checker. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.
- [34] Stathis Maneas, Kaveh Mahdavian, Tim Emami, and Bianca Schroeder. A Study of SSD Reliability in Large Scale Enterprise Storage Deployments. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, 2020.
- [35] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2015.
- [36] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD Failures in Datacenters: What? When? And Why? In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR)*, 2016.
- [37] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S. Gunawi. IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC)*, 2019.
- [38] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, 2007.
- [39] Peter J. Rousseeuw and Mia Hubert. Robust Statistics for Outlier Detection. *WIREs Data Mining and Knowledge Discovery*, 2011.
- [40] Bianca Schroeder and Garth A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000

Hours Mean to You? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, 2007.

- [41] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.
- [42] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In *Proceedings of the 2009 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2009.
- [43] Scott Shadle. NAND Flash Media Management Through RAIN. https://www.micron.com/-/media/client/global/documents/products/technical-marketing-brief/brief_ssd_rain.pdf, 2011.
- [44] Charles Spearman. The Proof and Measurement of Association Between Two Things. *American Journal of Psychology*, 100(3/4):441–471, 1987.
- [45] Amy Tai, Andrew Kryczka, Shobhit O. Kanaujia, Kyle Jamieson, Michael J. Freedman, and Asaf Cidon. Who’s Afraid of Uncorrectable Bit Errors? Online Recovery of Flash Errors with Distributed Redundancy. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC)*, 2019.
- [46] Guanying Wu and Xubin He. Delta-FTL: Improving SSD Lifetime via Exploiting Content Locality. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.
- [47] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. Lessons and Actions: What We Learned from 10K SSD-Related Storage System Failures. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC)*, 2019.
- [48] Qiao Zhang, Guo Yu, Chuanxiong Guo, Yingnong Dang, Nick Swanson, Xincheng Yang, Randolph Yao, Murali Chintalapati, Arvind Krishnamurthy, and Thomas Anderson. Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.
- [49] Xuebin Zhang, Jiangpeng Li, Hao Wang, Kai Zhao, and Tong Zhang. Reducing Solid-State Storage Device Write Stress through Opportunistic In-place Delta Compression. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.
- [50] Kai Zhao, Wenzhe Zhao, Hongbin Sun, Xiaodong Zhang, Nanning Zheng, and Tong Zhang. LDPC-in-SSD: Making Advanced Error Correction Codes Work Effectively in Solid State Drives. In *Proceedings of the*

11th USENIX Conference on File and Storage Technologies (FAST), 2013.

A Artifact Appendix

Abstract

The artifact consists of the first large-scale public dataset on real-world operational data of NVMe SSD. With this dataset, we have identified a series of major reliability changes in NVMe SSD. The community could leverage our dataset and findings to understand the major reliability changes in NVMe SSD, and design effective reliability solutions (e.g., detecting and predicting failures) in production environments.

Scope

Most major findings in the main text (i.e., Findings 1-8 and 10) could be validated by exploring the dataset. Moreover, practitioners could make use of this dataset to investigate the *fail-stop* and *fail-slow* failure characteristics of NVMe SSD. For example, the dataset could be used to design fail-slow detection algorithms or to predict fail-stop or fail-slow failure occurrences in large storage systems.

Contents

The dataset primarily covers:

- **SMART logs and failure tickets** of around 700K NVMe SSDs of 11 drive families from three vendors during a one-year span. Practitioners could make use of them to investigate the *fail-stop* failure characteristics of NVMe SSD.
- **Performance logs** (i.e., device-level write latency time series) of around 97K NVMe SSDs and 141K SATA HDDs. Practitioners could make use of them to investigate the *fail-slow* failure characteristics of NVMe SSD, and compare them with those of SATA HDD.

Hosting

The open-source dataset is hosted by Tianchi of Alibaba Cloud at <https://tianchi.aliyun.com/dataset/dataDetail?dataId=128972> with detailed instructions. Please refer to the above link for more information. We commit to ensuring the availability of this dataset.

CacheSack: Admission Optimization for Google Datacenter Flash Caches

Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister

Google Inc.

{twyang, pollen, uysal, aamerchant, wolfmeister}@google.com

Abstract

This paper describes the algorithm, implementation, and deployment experience of CacheSack, the admission algorithm for Google datacenter flash caches. CacheSack minimizes the dominant costs of Google’s datacenter flash caches: disk IO and flash footprint. CacheSack partitions cache traffic into disjoint categories, analyzes the observed cache benefit of each subset, and formulates a knapsack problem to assign the optimal admission policy to each subset. Prior to this work, Google datacenter flash cache admission policies were optimized manually, with most caches using the Lazy Adaptive Replacement Cache (LARC) algorithm. Production experiments showed that CacheSack significantly outperforms the prior static admission policies for a 6.5% improvement of the total operational cost, as well as significant improvements in disk reads and flash wearout.

1 Introduction

Colossus Flash Cache (Figure 1) is the general-purpose flash cache service for Colossus [20], the successor to the Google File System [19]. Disk reads are expensive and are a major cost in datacenters: while disks are growing in storage capacity, the IO capacity (the ability to offer disk accesses per second, mainly disk reads) is not growing proportionally. As a result, to provision the IO requirements, we need to deploy a lot of hard disks that are not for storage but for the target IO capacity, which is costly.

Colossus Flash Cache provides a cost-effective way to improve IO capacity while costing a fraction of an equivalent RAM cache or deploying more hard disks. The primary design goal of Colossus Flash Cache is to reduce the amount of hard disk reads, in order to reduce disk IO requirements and costs.¹ Colossus Flash Cache serves the read traffic of many widely-used Google services including Colossus and database

¹While reducing read latency is also a desirable goal, it is not a design goal for Colossus Flash Cache, and beyond the scope of this paper.

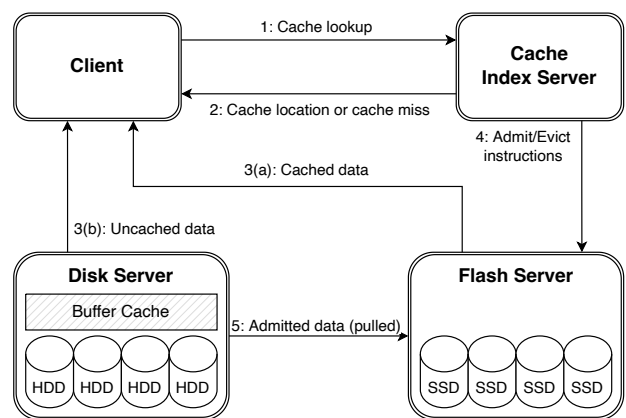


Figure 1: Colossus Flash Cache system.

systems such as BigQuery [37], BigTable [11], F1 [34], and Spanner [13].

CacheSack is the cache admission algorithm used by Colossus Flash Cache, intended to minimize the total cost of ownership (TCO). Compared to a RAM cache, a flash cache usually provides a much larger cache-to-storage capacity, and so a simple algorithm such as LRU may achieve a good cache hit-ratio. An idealized LRU is difficult to implement in a flash cache; we address this issue in Section 4. However, flash memory has limited write endurance, so may cause premature flash wearout and increase TCO. Write amplification and flash wearout, along with caching in Colossus disk servers, form a special challenge for designing a cache algorithm for Colossus Flash Cache.

2 Our contributions

CacheSack is the cache admission algorithm for Colossus Flash Cache, the successor to LARC (Lazy Adaptive Replacement Cache). CacheSack dynamically analyzes the cacheability of a workload and the given cache size, making the admis-

sion decision for the workload. CacheSack was deployed in Colossus Flash Cache in May 2021 and is now Colossus Flash Cache’s default cache admission algorithm. Our contributions are summarized as follows:

- CacheSack partitions traffic into multiple categories, estimates the disk reads and cost of write of each category, and formulates a knapsack problem that finds the optimal admission policy per category to minimize the overall cost, including disk reads and bytes written to flash.
- CacheSack effectively reduces the total cost of ownership (TCO) of Colossus Flash Cache. Compared to LARC, it results in 6% lower disk reads, reduces bytes written to flash by 26%, and improves TCO by 6.5% (one week average).
- CacheSack runs in real time, using a fraction of the resources of a cache index server.
- CacheSack is fully decentralized (as is Colossus Flash Cache). It requires only the information received by a single cache index server, and the failure of a single cache index server does not impact others.
- CacheSack supports major Google database systems and requires zero configuration if using these systems. For other applications, users only need to provide category annotations (Section 5.1).

3 Background

3.1 Write amplification

Non-sequential writes to a flash drive can cause serious write amplification [29, 42], a phenomenon where one logical write causes multiple physical writes. A flash byte has to be erased before it can be rewritten. A flash block is a continuous region of bytes in a flash drive and is the smallest unit that can be erased. As a result, a flash drive needs to move the live bytes in a flash block somewhere else before this flash block can be erased, which is called write amplification. Extra writes caused by write amplification reduce the IO performance and the lifetime of a flash drive; both greatly increase the cost of operating a flash cache.

Sequential cache evictions (like FIFO) result in large sequential areas that can be easily erased and reused later when admitting new data. By contrast, non-sequential evictions (such as those caused by LRU) result in a fragmented cache space and the flash drive has to move the interspersed live bytes somewhere else before erasing a block.

As a result, most existing eviction algorithms for RAM caches cannot be directly applied to flash caches, and write amplification is one of the most important factors to consider when designing a flash cache algorithm. Both Google [1] and Facebook [18] use FIFO-based evictions or other special purpose algorithms [36, 44] for production flash caches because of write amplification. Colossus Flash Cache reduces write amplification brought by non-sequential evictions by using

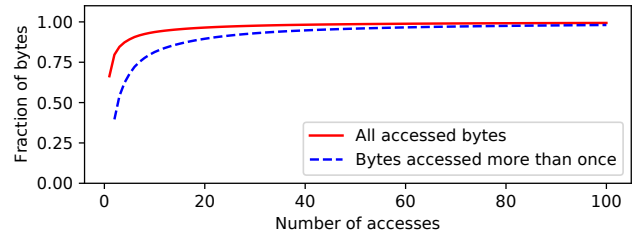


Figure 2: Fraction of bytes accessed a given number of times from disk plus flash over a week (right truncated at 100 accesses).

approximate LRU (Section 4).

3.2 Write endurance

Flash has limited write endurance, and thus admitting all data into Colossus Flash Cache upon write or even upon the first read would wear out the flash too soon, significantly increasing TCO. To mitigate this issue, Colossus Flash Cache previously used Lazy Adaptive Replacement Cache (LARC) [21] to exclude data that are accessed only once by inserting data at the second access. Figure 2 shows that more than 60% of the traffic of Colossus Flash Cache is accessed only once, and so LARC can greatly reduce bytes written to flash and avoid cache pollution.

However, excessive flash writes are still possible with LARC, and as a workaround, Colossus Flash Cache used a write rate limiter to avoid an excessively high write rate. This is, however, a blunt approach, since it does not accurately factor in the impact on overall cost, and treats all workloads similarly. It may be preferable to allow some highly cacheable workloads to burst writes at the expense of other less cacheable workloads rather than throttling all write rates. CacheSack uses a more flexible and accurate approach by optimizing the total costs, including the write costs and the cost of disk reads.

3.3 Capturing second-access hits

LARC leverages the fact that a large fraction of data is accessed only once. Inserting data into cache only upon the second access avoids flash writes for data accessed once, reducing flash wear. However, the cost is that all second accesses are cache misses. Figure 2 shows that of the data accessed more than once in our workloads, 39% is accessed exactly twice, and these second accesses are cache misses under LARC. This has a significant performance impact, as was also observed in Facebook’s cache for social network photos [36]. Our workaround for this when using LARC was to monitor the performance loss, and to manually turn off LARC (i.e., admit all data on the first miss) for workloads that suffered a significant performance penalty. However, the

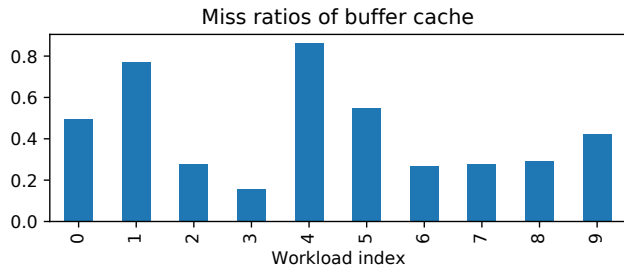


Figure 3: Miss ratios for 10 workloads at the disk server buffer cache if there is no Colossus Flash Cache (simulated).

manual maintenance to identify and set up special cases became more and more labor-intensive with the rapid adoption of Colossus Flash Cache in production. In our redesign, it was a requirement that the cache admission algorithm should be automatic and not require manual adjustments.

3.4 Colossus buffer cache

In addition to Colossus Flash Cache, Colossus maintains a RAM *buffer cache* in the lower level disk servers that buffers recent reads and writes as well as data prefetched from the disk. A cache miss in Colossus Flash Cache does not cause a disk read if the access hits in the buffer cache. Many Colossus workloads use the buffer cache extensively to improve IO performance.

In many cases, the cache hit ratio of Colossus Flash Cache is only weakly correlated with the actual disk read reduction, especially for workloads that are highly optimized for the buffer cache. Figure 3 shows simulated miss ratios of the disk server buffer caches with no Colossus Flash Cache for ten selected workloads, and they range from below 20% to over 80%. These miss ratios represent the upper bound on how far Colossus Flash Cache can improve the disk read rates. For workloads with low buffer cache miss ratios, hits in Colossus Flash Cache may simply replace buffer cache hits without improving the disk read rates. As a result, flash cache hit ratios are not a good metric to measure the efficacy of Colossus Flash Cache. In fact, our production results (Section 7.1) show that an admission policy can sometimes provide a higher hit ratio in Colossus Flash Cache but cause worse disk read rates.

3.5 Online and realtime requirements

Colossus Flash Cache is a fully decentralized system, so its cache algorithm can only use the resources of individual cache index servers, and heavyweight algorithms, such as machine learning (ML) models, may not be feasible. The binary of Colossus Flash Cache is updated on a weekly basis, while workloads change much more rapidly, so it is difficult for an offline-trained static model updated with the binary to adapt

to workload changes. Therefore, we decided to use an online-trained model.

4 Overview of Colossus Flash Cache

Colossus Flash Cache consists of independent cache index servers. A cache index server does not directly hold cached data, but keeps an in-memory lookup table, called the index, that tracks the locations of cached data stored in the flash drives that reside on independent storage servers.

When a Colossus Flash Cache client requests to access data stored in Colossus, the client first sends an RPC to a Colossus Flash Cache cache index server (see Figure 1) to determine if the requested data are already cached on a flash server (a flash hit). If so, the cache index server sends back sufficient information for the client to access the flash copy of the data directly from the flash server. For a flash cache miss, the client contacts the disk server to read the data, while the cache index server independently decides whether to admit the data into the flash cache. If the cache index server decides to admit the data into flash, it instructs the flash server to pull the data from the disk server directly. The extra latency of communicating with the cache index servers is negligible compared to typical remote disk read latencies, and the latencies between remote flash reads and remote disk reads are in different orders of magnitude, so Colossus Flash Cache typically reduces overall latency, although this is not an explicit service goal. The goal is reducing TCO by avoiding expensive disk reads.

The *buffer cache* of a disk server also caches recently accessed data and prefetches a small amount of data into memory for a few seconds, so that reading recently-accessed data from an disk server does not necessarily cost extra disk reads. Colossus users are encouraged to design their workloads to improve IO performance by utilizing this buffer cache.

Colossus Flash Cache uses an *approximate* LRU eviction strategy to manage evictions. An idealized LRU cache would always evict the least recently used block from the cache when the cache is full. However, idealized LRU evictions cause non-sequential writes to flash, resulting in write amplification [29, 42]. To mitigate the issue of write amplification, Colossus Flash Cache uses evictions similar to Second Chance [30] to approximate LRU evictions: each Colossus Flash Cache cache index server manages a FIFO queue of many fixed-sized Colossus files (typically 1 GiB), each of which contains cache blocks. When evicting the file from the tail of the queue, we reinsert 28% of the most-recently-used blocks into the file at the head of queue. The percentage of the reinserted blocks is a tradeoff between the amount of hot blocks recycled, which improves the cache hit ratio, and the rate of reinsertion into flash, which increases write amplification. The current value (28%) is selected experimentally to strike a good balance between cache performance and write amplification. This way, the write amplification factor is effectively 1.28. A comparison of the performance of Second

Chance [30] indicates that the performance is quite close to that of LRU. Therefore, for ease of modeling, we approximate Colossus Flash Cache as an LRU cache.

Each Colossus Flash Cache server maintains a *ghost cache* [21], an in-memory lookup table that maps the key of data to the data’s last access time, regardless of whether they are actually cached on flash. This is a key component of CacheSack, which relies on inter-arrival times to quickly build all the estimates described in Section 5.

Each cache index server represents a fraction of the key space, and one server’s failure does not impact other cache index servers. To maintain the same reliability, CacheSack is also designed in the same decentralized manner: each cache index server runs its own CacheSack model, using only the information received by the cache index server, and its admission decisions do not affect other cache index servers.

5 CacheSack

5.1 Traffic partitioning

CacheSack partitions potential cache blocks into many *categories*, and assigns an admission policy to each category.

The majority of Colossus Flash Cache traffic comes from Google’s database systems like BigTable and Spanner where categories can be well-defined. For database traffic, CacheSack defines a category as the combination of the table name, locality group [11, 13], and type for BigTable and Spanner, and a similar combination for other databases. Since Colossus Flash Cache is also available for other Colossus users, those users can define their own categories by annotating their data. If a user does not provide a category annotation, CacheSack will use the user name contained in the Colossus file path.

CacheSack then selects the right policy based on the pattern that category exhibits. Later, we will explain how we formulate CacheSack as a knapsack-like problem: given the cache capacity, how CacheSack chooses the items (categories) to minimize the overall cost.

5.2 Admission policies

We consider four admission policies that can be assigned to each category:

- **AdmitOnWrite**: Inserts a cache block at a write access or on any read cache miss.
- **AdmitOnMiss**: Inserts a cache block on any read cache miss, but does not insert a block at a write access. This is the conventional admission policy used in most of the cache literature.
- **AdmitOnSecondMiss (LARC)**: Equivalent to Lazy Adaptive Replacement Cache (LARC); Inserts a block only after the second read access (miss), and only if the last access time is not older than the oldest last access time of the blocks in the cache, to reduce the insertion

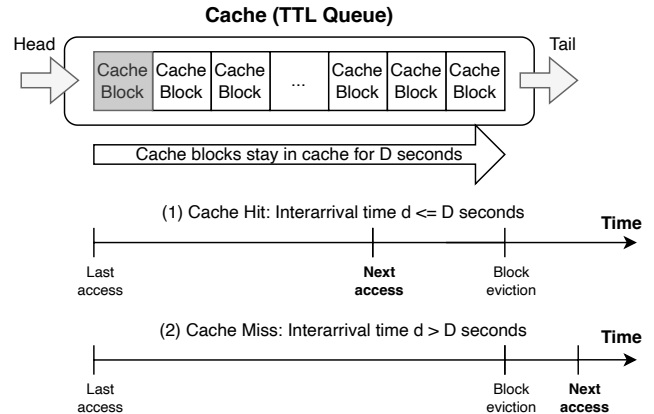


Figure 4: LRU evictions are approximated by TTL evictions with the modeled retention time D while the TTL counter of a cache block is reset whenever the cache block is accessed. If the access interarrival time d is less than or equal to D , this access is a cache hit and we move the cache block to the head of the queue (the TTL counter is reset). If the access interarrival time d is greater than D , this access is a cache miss and we insert the cache block to the head of the queue (the TTL counter is also reset).

rate of cold blocks. LARC is scan resistant: any scanned data (accessed exactly once) will not be admitted.

- **NeverAdmit**: Never inserts blocks.

We can sort these policies by aggressiveness: $\text{NeverAdmit} < \text{AdmitOnSecondMiss} < \text{AdmitOnMiss} < \text{AdmitOnWrite}$.

5.3 Fast approximation to an LRU model

To determine the best policy for the cache, the most intuitive way is to simulate all possible policy-category combinations, which is a combinatorial knapsack problem (NP-Hard). Because CacheSack currently allows up to 5000 categories (Section 6.1) and uses 4 policies, there are up to 4^{5000} combinations and the knapsack problem can not be done even with downsampled traces. Instead, we use a fast approximation for modeling an LRU cache, by introducing the *modeled cache retention time*. The cache retention time is the maximum duration that a block stays in the LRU cache without any intervening accesses to it. In practice, the cache retention time varies slowly over time. Here we assume the *modeled cache retention time* is a constant D and this assumption will make all our estimates just approximations.

We use **AdmitOnMiss** as an example. For a given block, when a read access arrives, we can compute d , the time since last access (which is ∞ if the current access is the first read). We can classify the inter-arrival times by using D (Figure 4):

- $d \leq D$: An access arrives before the block leaves the cache, and therefore the access generates a cache hit and moves the block to the head of the queue.

- $d > D$: An access arrives after the block leaves the cache, and therefore it is a cache miss, which causes a write to the cache.

In other words, we approximate the LRU cache by a cache that has the TTL value D and resets the TTL counter of a block when receiving an access to the block. The theoretical aspect of the TTL approximation was also studied in literature. Fagin [17] showed the TTL approximation is asymptotically exact for independent and identically distributed requests, and [23] proved that given the assumption that data accesses are stationary and ergodic, the TTL approximation will converge to an LRU cache as the cache size goes to infinity. The accuracy of the TTL approximation in production is analyzed in Section 7.1.

A cache miss in Colossus Flash Cache will cause a disk read if it is also a miss in the Colossus buffer cache. Each cache index server maintains a *buffer cache simulator*, and when $d > D$, we run the simulator and see whether it is a miss.

This way, when a new access arrives, we are able to update the disk reads, cache usage, and bytes written to flash cache caused by admitting the block using `AdmitOnMiss`. We can also compute the same quantities for other policies: `AdmitOnSecondMiss`, `AdmitOnWrite`, and `NeverAdmit`. The detailed estimation is described in Appendix A.2.

A nice property of this approximation is that the estimates for a block are not affected by other blocks or policies, as long as the modeled cache retention time is given. Therefore, the disk reads, cache usage, and written bytes caused by admitting a category are just the sums of the corresponding block-level quantities.

5.4 Knapsack problem

Once we have the estimates for disk reads, cache usage, and bytes written to flash cache for each policy-category pair, we have a knapsack problem: find the optimal policy per category to minimize the overall cost (disk reads and written bytes) while fitting within the cache. We omit the definitions of the relative cost of disk reads, bytes written to flash, and flash storage, because they are confidential.

We further allow *fractional* policies: CacheSack can apply a policy to a fraction of a category. For example, CacheSack may decide it is optimal to apply `AdmitOnMiss`, `AdmitOnSecondMiss`, `AdmitOnWrite`, and `NeverAdmit` to 30%, 20%, 10%, and 40% of blocks in a category, respectively. Then the problem becomes a *fractional* knapsack problem [14] that finds the optimal policy fractions per category to minimize the overall cost. The advantage of considering a fractional knapsack is that it can be solved efficiently by a greedy algorithm, as opposed to a combinatorial knapsack that is NP-Hard. Our problem is slightly different from the original fractional knapsack in [14] because we need to decide four fractions per category instead of two. Appendix A.4

explains the details of how we solve our problem by a greedy algorithm after applying Andrew’s monotone chain convex hull algorithm [2]. We note that if an LRU cache is perfectly modeled by the TTL approximation, the resulting cache retention time of the LRU cache is exactly D after applying the optimal policy fractions per category.

5.5 Optimization over modeled cache retention times

The knapsack problem in Section 5.4 is to find the optimal policy fractions for a *given* modeled cache retention time D , which can not be known in advance. Thus, we need to solve the same knapsack problem for *all* possible D . To do this in production, we can have a set of predefined modeled cache retention times: $0 < D_1 < D_2 < \dots < D_m = \mathbf{D}$ where \mathbf{D} is a suitable upper bound, and solve m different knapsack problems. Thanks to the greedy algorithm, we can still solve many knapsack problems (currently 127, Section 6.2) quickly.

6 CacheSack in production

CacheSack is now deployed in production as the default cache admission algorithm for Colossus Flash Cache. This section explains the engineering efforts needed to do so.

6.1 Category assignment

The number of categories encountered in production can not be known in advance, so we balance the need for accuracy and space by hashing a category to one of 5000 buckets. Categories assigned to the same bucket are treated as combined in the optimization. The number of hash buckets is a trade-off between memory usage and hash collisions. The typical number of categories per server is less than 100 and our experiments showed that with 5000 buckets, 95% of the clients see a hash collision rate lower than 1% and the worst collision rate is less than 5%. Further, cache collisions are not persistent, since each cache index server uses a different hash key and changes it periodically to break possible spatial and temporal correlations.

A bucket without sufficient training data might not provide meaningful metrics. If a bucket contributes to less than 0.1% of total lookups, it will be aggregated to a single *catch-all* bucket before solving the knapsack problem.

6.2 Modeled cache retention times

Currently, CacheSack uses 127 predefined cache retention times: 15 minutes, 1.06×15 minutes, $1.06^2 \times 15$ minutes, ..., $1.06^{126} \times 15$ minutes ≈ 16 days; the 128th value is reserved for positive infinity.

These retention times are decided in the following way. We first determine the working range. A retention time less

than 15 minutes means we evict and insert cache blocks in an extremely aggressive way, which would cause serious flash wearout. By policy, any cache block is forced to leave the cache if it stays more than 15 days. Hence we set the modeled working range of retention times as 15 minutes to 15 days. We then decide the number of retention times to model. We tried 127 (6% geometric increase) and 255 (3% geometric increase) retention times, and our experiments showed that 127 retention times gave similar results while reducing RAM usage by half.

6.3 Ghost cache

Since LARC was Colossus Flash Cache’s previous admission control, a *ghost cache* was implemented in cache index servers. It is an in-memory lookup table that maps a data’s key to the data’s last read access time, and LARC uses the information to determine whether to admit the data on miss. CacheSack uses the same ghost cache to obtain inter-arrival times. In addition, to build the metric estimate for `AdmitOnWrite`, we expanded the ghost cache so that we know whether the last access is a write access. To build the metric estimate for `AdmitOnSecondMiss`, we use the ghost cache to record the most two recent access times.

Because the ghost cache is the ground truth for CacheSack, the ghost cache must contain sufficient history. The optimal solution of CacheSack will not be affected as long as the ghost cache TTL, the time since the oldest last access time of the blocks in the ghost cache, is greater than the optimal modeled cache retention time. As a rule of thumb, we provision the size of the ghost cache so that its TTL is at least twice the solved optimal modeled retention time (typically about four hours).

6.4 Buffer cache simulators

A cache miss in Colossus Flash Cache causes a disk read only if it is also a miss in the buffer cache. CacheSack simulates the buffer cache to determine whether the current miss in Colossus Flash Cache is also likely a miss in the buffer cache. In fact, we need *many* simulators: one for each pair of policy-retention time so there are 382 simulators ($3 \times 127 + 1$, the retention time does not affect `NeverAdmit`). Running the simulators is the most computationally intensive component in the CacheSack model. Fortunately, the buffer cache simulator is simple enough and only requires the access history in the past few seconds so it only moderately increases CPU load on the low-QPS servers (5% CPU usage).

6.5 Model training

We use a simple scheme to train the CacheSack model: the model is reset every 5 minutes and is trained based on the

lookups in this 5-minute period. We note that a lookup contains the access times of the most recent two accesses and therefore the lookups in a 5-minute period may contain the information of many hours.

The selection of the training duration is a trade-off. Using a larger training duration means the model can be improved by more training data and longer time horizon, while the model can react more quickly to changes in the workload with a shorter duration. We tested several training durations and found that 5-minute one gave most disk read reduction, although we did not find significant differences among all candidates.

6.6 Lessons learned

Automatic cache optimization incentivized user adoption

In deciding whether to use Colossus Flash Cache, users weigh both the likely TCO improvement and the engineering effort required to configure and maintain it. In the past, users had to manually choose the admission policy (using `AdmitOnMiss` or `AdmitOnSecondMiss`) based on knowledge of their workload or by running A/B experiments with the assistance of the Colossus Flash Cache team. For heavy users like Spanner, Colossus Flash Cache had to provide heuristic, hand-tuned admission policies to improve cache performance. Such human tuning and maintenance usually requires effort from both the users and the Colossus Flash Cache team, which can discourage the adoption of Colossus Flash Cache if the expected hardware resource saving does not justify the extra engineering cost.

We found that CacheSack greatly incentivized users to adopt Colossus Flash Cache. The automatic cache provisioning brought by CacheSack requires almost no configuration and maintenance so that it can be set and forgotten. We found that new users were more willing to use Colossus Flash Cache once they knew it would automatically adjust the cache policy based on their workloads.

Some of Colossus Flash Cache’s existing users have independently verified that CacheSack applied appropriate admission policies to their workloads, based on the knowledge of their workloads and reporting provided by Colossus Flash Cache. One user experimentally overrode CacheSack with manually optimized policies and found that CacheSack worked as well as manual policy tuning. After CacheSack became the default admission policy in Colossus Flash Cache, we were able to retire the hand-tuned optimization for Spanner, and our existing users did not need to manually adjust the policy anymore.

Experiment infrastructure accelerated feature development

The development of CacheSack was significantly benefited by the experiment infrastructure of Colossus Flash Cache.

The experiment infrastructure allows developers to test new features by using 10% of the cache index servers, and because cache index servers are independent and isolated, any experiment can only cause minor service degradation in the worst case. Before the full deployment, we ran CacheSack as an experiment for a few months and most of the issues were identified and corrected during the experimental phase. In fact, there was no binary rollback caused by CacheSack since the full deployment.

In addition, because each server represents a fraction of the key space, which is permuted randomly, each server is statistically indistinguishable. We can have simultaneous comparisons between CacheSack and the control group to see whether CacheSack works as expected and identify any issues. The experiment infrastructure is extensively used by the developers of Colossus Flash Cache for new features, and the impact of a new feature can be accurately measured before the full deployment.

Model introspectability and maintainability played important roles

We found that the model introspectability played an important role for the adoption of the new cache algorithm. Because any cache algorithm of Colossus Flash Cache will be operated and maintained by developers and site reliability engineers (SREs) after the initial deployment, one requirement of deploying a new cache algorithm is that the model behavior can be fully understood and monitored by the developers and SREs. CacheSack satisfies this requirement as it only assumes that the TTL approximation (Section 5.3) is sufficiently close to the eviction of Colossus Flash Cache, and all model behaviors can be derived from this assumption. Another advantage of a highly introspectable model is that the developers (besides the original designers) of Colossus Flash Cache can easily ensure thorough test coverage, validate software releases, add extend the original functionality of CacheSack without assistance from the original designers. After the deployment of the original CacheSack, it became the foundation of further optimizations for Colossus Flash Cache.

It is also worth mentioning that CacheSack is simple enough to be implemented by limited extensions to the original codebase of Colossus Flash Cache. In particular, the optimization was implemented as a simple greedy algorithm instead of using a generic linear program solver library. This did cost extra time for development, but we decided to do so because it allowed us to minimize the computational overhead and increase system reliability by reducing external dependencies. More importantly, anyone familiar with the ecosystem of Colossus Flash Cache can easily maintain CacheSack or develop new features based on it. The implementation of CacheSack can evolve continuously with Colossus Flash Cache, reducing maintenance burden. Since the completion of the initial deployment, both maintenance and new feature devel-

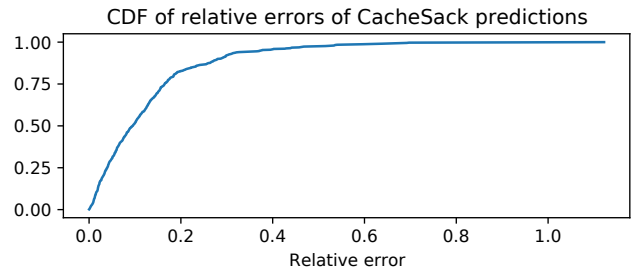


Figure 5: CacheSack disk read rate prediction errors relative to the actual value in production (CDF).

opments have been completely handled by Colossus Flash Cache developers and SREs without the need for involvement from the original designers.

7 Evaluation

7.1 Production evaluation

Model accuracy

There are two LRU approximations in Colossus Flash Cache: Colossus Flash Cache uses Second-Chance-like approach to approximate LRU evictions (Section 4), and CacheSack models an LRU cache as a TTL approximation (Section 5.3). Therefore, it is important to verify that the CacheSack model is a good enough approximation to the actual Colossus Flash Cache. We examined the accuracy of CacheSack in the following way. For each client, the solution to the knapsack problem in Section 5.4 gives the predicted disk reads when using the optimal admission policies. Then Colossus Flash Cache applies the optimal policies in production so we compared the predicted disk reads with the actual disk reads. Figure 5 shows the prediction errors of CacheSack relative to the actual values obtained from the disk servers; 51% of the relative errors are within 10% and 82% of the relative errors are within 20%.

Policy distribution

Figure 6 shows the policy distributions suggested by CacheSack in the selected datacenters of various workloads. We can see that each datacenter has a different workload pattern and CacheSack adaptively decides suitable admission policies based on workloads and cache sizes. Although it would be possible for manual selection of static policies to match each datacenter workload, CacheSack is able to reduce the human toil, response delay, and operational complexity required to maintain these assignments.

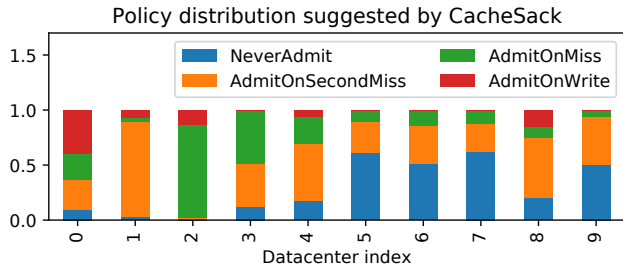


Figure 6: Policy distribution suggested by CacheSack in selected datacenters, demonstrating a variety of workload responses.

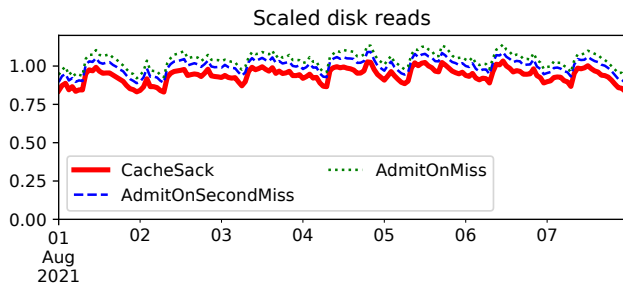


Figure 7: Disk reads of different admission policies in production, divided by the average value for AdmitOnSecondMiss.

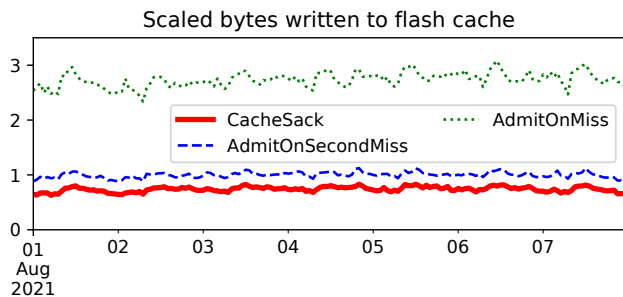


Figure 8: Written bytes of different admission policies in production, divided by the average value for AdmitOnSecondMiss.

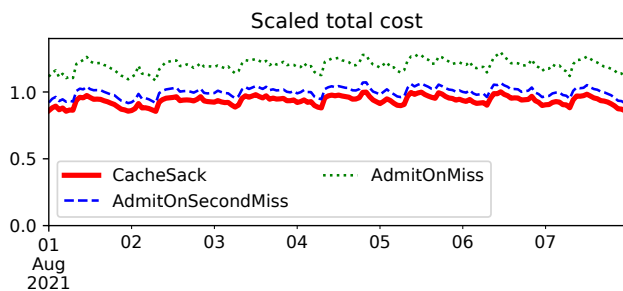


Figure 9: Total cost (a function of disk reads, flash storage and written bytes) of different admission policies in production, divided by the average value for AdmitOnSecondMiss.

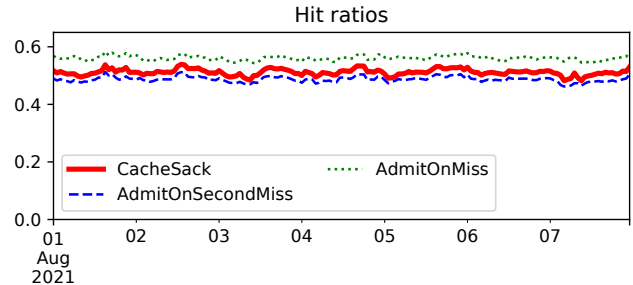


Figure 10: Hit ratios in Colossus Flash Cache of different admission policies in production.

Production experiments

By using the experiment infrastructure of Colossus Flash Cache, we can compare the performance of different cache algorithms in production. Because each cache index server represents a fraction of the key space, the pattern of workload each cache index server receives is statistically indistinguishable. We let 10% of the cache index servers run static AdmitOnMiss and another 10% of the cache index servers run AdmitOnSecondMiss so that we can compare CacheSack, static AdmitOnMiss and static AdmitOnSecondMiss simultaneously in production.

From Figure 7 and 8 we see that compared to AdmitOnSecondMiss, CacheSack results in fewer disk reads (6% of one week average) and reduces 26% (one week average) written bytes to flash, and Figure 9 shows that CacheSack effectively reduces the total operating cost in production: the cost of disk reads, flash cache writes and flash storage of CacheSack is 93% of AdmitOnSecondMiss and is 78% of AdmitOnMiss (one week average).

Figure 10 shows that CacheSack has a higher hit ratio than AdmitOnSecondMiss but lower than AdmitOnMiss. Nevertheless, AdmitOnMiss is not the best choice. Figure 7 shows that AdmitOnMiss has the worst disk read reduction even though it has the highest hit ratio. Because of the lower-level buffer cache, a higher hit ratio in the flash cache does not necessarily imply fewer disk reads: many major Colossus users optimize their workloads by accessing the same data many times within the first few seconds so that only the first access causes an actual disk read. In this case, AdmitOnMiss generates many hits that do not reduce disk reads at all. AdmitOnSecondMiss resolves this issue by avoiding a cache insertion if the most recent access time is too recent to expect that the data has left the buffer cache.

7.2 Evaluation by simulations

In addition to production experiments, we also used the *Colossus Flash Cache simulator* to test the performance of CacheSack in a variety of configurations and contexts, such as

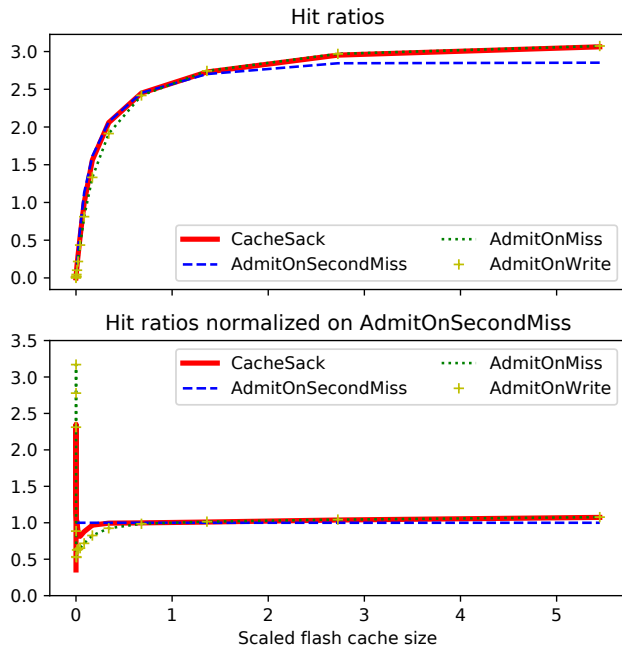


Figure 11: Hit ratios in Colossus Flash Cache of different admission policies in simulation. Above: Original hit ratios. Below: Values relative to AdmitOnSecondMiss.

cache size and optimization iteration period. The Colossus Flash Cache simulator is used for multiple purposes including performance-regression testing by Colossus Flash Cache developers and for datacenter resource planning by Colossus Flash Cache clients. The Colossus Flash Cache simulator uses the same production code as Colossus Flash Cache and we use production traces as the input of the simulator.

We first compare the performance of CacheSack, to the static admission policies AdmitOnMiss, AdmitOnSecondMiss and AdmitOnWrite for various cache sizes. We use here a two-day trace from one large (order of million QPS) production cache as a representative. This trace reflects a uniform sample of the data accesses from a large collection of internal production workloads.

Impact of cache size on performance

When the cache size is small, AdmitOnSecondMiss has a better performance than AdmitOnMiss or AdmitOnWrite because single-use keys are excluded. On the other hand, AdmitOnMiss and AdmitOnWrite will outperform AdmitOnSecondMiss for a large cache because the cache benefit of second accesses is gained.

CacheSack learns to use a more conservative policy for a small cache and a more aggressive policy for a large cache. Figure 11 and Figure 12 show that CacheSack can provide a good performance for the entire range of flash cache sizes.

It is also interesting to see the amount of written bytes

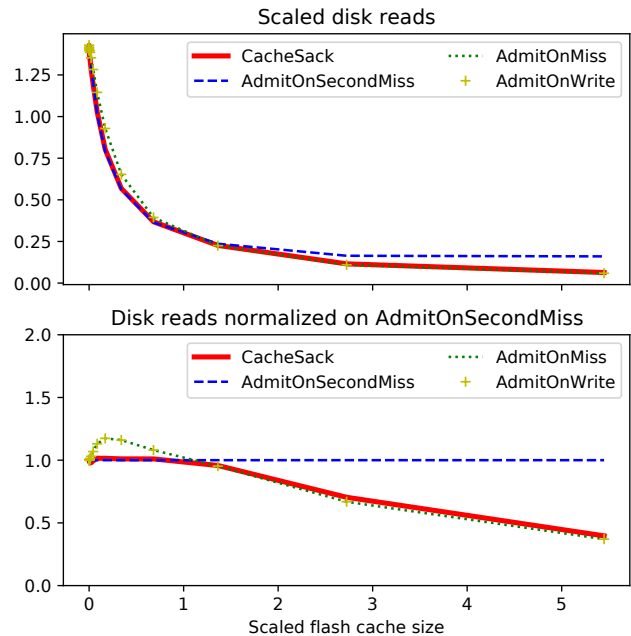


Figure 12: Disk reads of different admission policies in simulation. Above: Constant scaling by dividing the values by the average value for AdmitOnSecondMiss. Below: Values relative to AdmitOnSecondMiss.

caused by different admission policies in Figure 14. For AdmitOnMiss, AdmitOnWrite and AdmitOnSecondMiss with excessively small cache, blocks are frequently evicted from and reinserted into the cache, resulting in a very large amount of written bytes, especially for AdmitOnMiss and AdmitOnWrite. CacheSack, on the other hand, takes into account the cost of written bytes, and therefore only admits the most valuable part of the workload into the cache.

We can also view the total cost (a confidential function of disk reads, flash storage, and written bytes) as a function of cache size. When the cache size is small, disk reads and writes to flash are the largest contributions to cost, while flash storage is the largest cost component for larger cache sizes. Therefore, the total cost is a U-shape curve and we are able to find the optimal cache size that minimizes the total cost. Figure 13 shows that CacheSack gives the lowest total cost for all cache sizes. CacheSack avoids the trade-off and provides robust good behavior over the range of cash sizes.

Optimization frequency

We evaluated the system performance on the choice of different optimization frequencies. Here we test different lengths of training duration from one minute to eight hours, which span a majority of the observed time variation of workloads. Figure 15 shows that the training duration does not significantly impact the performance and all the cost metrics are

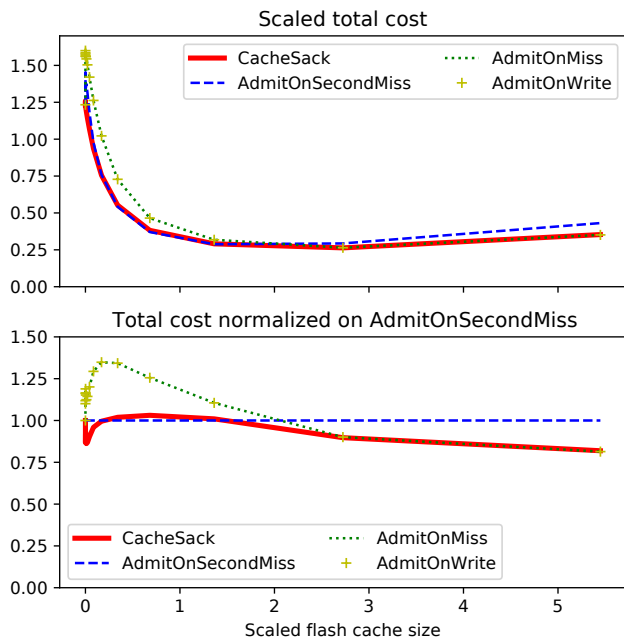


Figure 13: Total cost (a function of disk reads, flash storage and written bytes) of different admission policies in simulation. Above: Constant scaling by dividing the values by the average value for AdmitOnSecondMiss. Below: Values relative to AdmitOnSecondMiss.

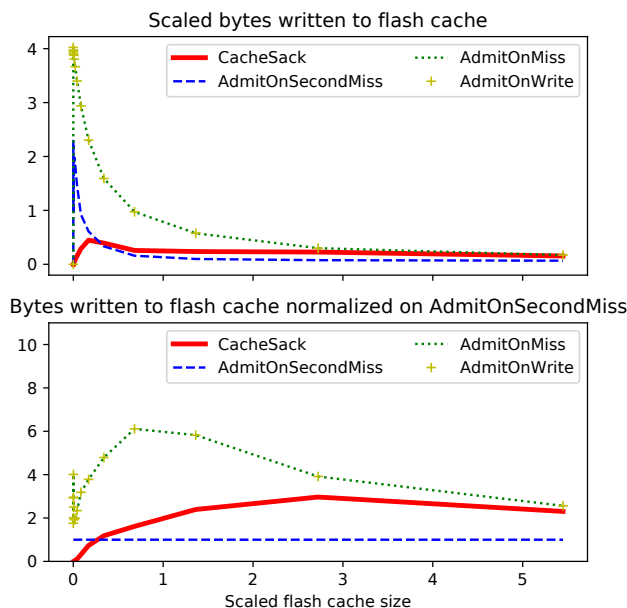


Figure 14: Written bytes of different admission policies in simulation. Above: Constant scaling by dividing the values by the average value for AdmitOnSecondMiss. Below: Values relative to AdmitOnSecondMiss.

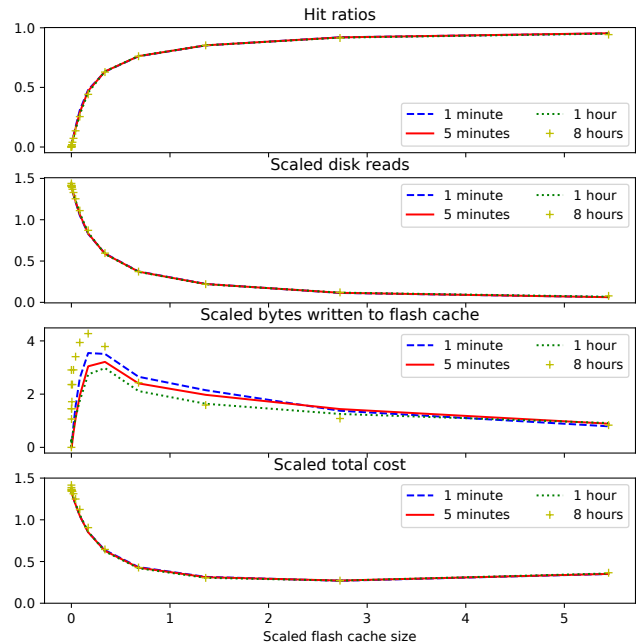


Figure 15: Hit ratios, disk reads, written bytes, and total cost (a function of disk reads, flash storage and written bytes) of Colossus Flash Cache with different training durations. Disk reads, written bytes, and total cost are divided by the average value for the 5-minute training duration.

similar. Because this method is insensitive to this parameter, customized or automated tuning was deemed unneeded, and the entire deployment currently uses a single value.

8 Related work

Production flash cache algorithms

Both Google [1] and Facebook [18] were using FIFO-based evictions in their production flash caches to trade cache performance for managed write amplification. RIPQ [36] is a non-FIFO, advanced flash cache algorithm that brings higher hit ratios while write amplification is well-control. Flashfield [16] further improves RIPQ's write amplification by using DRAM as a buffer, and only writes flash-worthy objects into flash, predicted by a lightweight support vector machine classifier. CacheLib [7] resolved Flashfield's issue that the TTLs of objects in the DRAM buffer are too short to be effective. CacheLib uses Bloom filters to count the number of accesses per object in the past six hours (similar to TinyLFU [15]), to predict the number of accesses in the future, and use FIFO for eviction. Kangaroo [26] further improves CacheLib's performance for tiny objects. DSS [28] uses predefined rules to classify I/O requests into different priorities, and applies heuristic admission and eviction policies to different priorities. DSS has been implemented in Intel's Cache Acceleration

Software. Amazon’s AQUA [3], which is conceptually similar to CacheSack, analyzes workload patterns to place data into the appropriate tier.

CacheSack’s high-level idea is similar to Flashfield and CacheLib: keep the eviction simple to control write amplification, and use a more sophisticated admission to improve cache performance and flash write endurance. For evictions, Flashfield uses the CLOCK [12] approach and CacheSack uses Second Chance [30] to achieve LRU-style evictions. On the admission side, instead of using DRAM as a buffer, CacheSack has no in-memory buffer and expands the metadata table (ghost cache) for a more complete history; the median of the ghost cache TTL is 20 hours, which is several times longer than the information used by Flashfield and CacheLib. With a more complete history, CacheSack is able to build a more sophisticated model for admission. CacheSack also considers the two major costs of operating flash caches, disk reads and flash wearout, as a whole, and minimizes the overall operation cost.

CacheSack also utilizes the advantage that the categories are well-defined in the database systems served by Colossus Flash Cache. Classifying unstructured data is usually a difficult problem in machine learning. For Google’s database systems, the classification is naturally given, and the categories often hint their cacheability.

Admission algorithms

LARC [21] was previously used by Colossus Flash Cache as the default admission policy. LARC is designed for flash caches, and reduces write rate by inserting an object into the cache only when it is read a second time, based on the observation that most objects are read only once. Thus, inserting only the objects that are read a second time into the cache significantly reduces the write rate and the cache pollution. This strategy is particularly useful when a significant portion of the traffic is accessed only once, for example, Tencent’s photo traffic [41], and AliCloud [24]. However, LARC loses all the first cache hits and becomes undesirable for long-tail accesses like Facebook’s cache for social network photos. In the past, Colossus Flash Cache disabled LARC for workloads in which LARC underperformed. Selective admissions like TinyLFU [15] (non-window version) and HEC [42] that sacrifice the first few hits to determine the cacheability of data likely have the same issue.

TinyLFU [15] works by comparing the expected hit ratio of a newly accessed object against that of the object that would be evicted next from the cache, inserting the new object if its likely hit ratio is higher. Any eviction policy can be used to select the eviction victim (LRU is typical). TinyLFU predicts hit ratios for the objects using approximate counting (Bloom filters) of access frequency. TinyLFU also needs some extra structures to work properly: *Doorkeeper* is used to filter one-accessed blocks (the same use of LARC’s ghost cache), and a

DRAM buffer cache in front of TinyLFU (W-TinyLFU). All these structures require extra parameter tuning, which does not best fit the needs of Colossus Flash Cache as a general-purpose cache. mARC [32] uses ARC [27] as the eviction policy and dynamically determines whether to admit data on the first miss (naive ARC) or second miss (LARC). UBC [31] proposes a low-overhead mechanism to partition shared on-chip cache.

Eviction algorithms

There are also extensive studies on advanced eviction algorithms. Beckmann and Sanchez’s method [5] evicts a block based on the block’s economic value added. Instead of LRU or LFU that require specific data structures, Hyperbolic Caching [9] evicts a block based on a time-decay (hyperbolic) value function and uses a sampling technique to resolve the issue of the data structure requirement. Similarly, LHD [4] evicts the block of the lowest hit density, the number of hits per cache byte-second, and also applies a sampling technique to overcome the data structure issue. Hawkeye [22] assumes that the recent history can predict the near future and hence one can train a predictor learned from Belady’s OPT [6] running on the recent traces. [40] considers an ensemble of candidates, which can be a set of existing algorithms, or the same algorithm with different parameters, runs scaled-down simulations on each candidate and periodically adopts the most performant one.

Machine learning algorithms

With the recent rapid development of machine learning (ML), there are also a few papers that adopt ML techniques to enhance cache performance. LFO [8] and LRB [35] use ML models to learn Belady’s OPT [6], and apply the ML models to CDN (Content Delivery Networks) caches. Parrot [25] also use ML to learn Belady’s OPT from history, but uses modern deep learning architectures like Transformer [38] and BiDAF [33]. [41] utilizes a concept similar to LARC [21] that the majority of traffic is accessed just once, and uses ML models to predict whether data is worth inserting into the flash cache. The algorithm showed a large flash write reduction in Tencent’s photo cache system as well as the improvement of hit ratios and latency. LeCaR [39] uses an ML approach to adaptively decide the better policy between LRU and LFU at eviction time. Zhou and Maas [43] model the inter-arrival times of a block as a log-normal distribution and learn the parameters from traces; then the evictions are executed in the manner of Belady’s OPT: the block with lowest probability to get the next access in the near future will be evicted.

9 Conclusions

In this paper, we introduce CacheSack, an admission policy optimization for Google’s datacenter flash caches. CacheSack provides an efficient estimation for the performance metrics of an LRU-style cache under various configuration options. We use a knapsack approach to identify the optimal admission policies to minimize the total cache-operating cost. We share the experience of deploying CacheSack in Colossus Flash Cache, the general-purpose flash cache serving Colossus, which has since become the default admission policy. CacheSack requires less manual configuration than the previously used cache admission algorithm (LARC), significantly reduces disk reads and bytes written to flash, and improves TCO by 6.5% compared to LARC.

Acknowledgements

The authors would like to thank Cory Casper, Junaid Khalid, Martin Maas, and Richard McDougall for their assistance in various stages of the design and implementation of this algorithm. We would also like to thank Dan Gibson, Larry Greenfield, Aaron Laursen, Milo Martin, Damodharan Rajalingam, and John Wilkes, as well as the anonymous reviewers and our conference shepherd, for their reviews and suggestions that improved this manuscript.

A Mathematical model of CacheSack

A.1 Model assumption

CacheSack models the cache as using LRU evictions. Colossus Flash Cache considers data for caching to be immutable after being written, i.e. the first access is a write, and subsequent ones are reads; mutability is handled by higher layers in the system. The CacheSack model does not need the immutability assumption, but we keep it to align with the actual system; the model can be easily modified for the mutable case.

A.2 Metric estimation of an LRU cache

We begin with `AdmitOnMiss`. For a given block b , let $t_1, t_2, t_3, \dots, t_n$ be the read access times, and $t_0 = -\infty$ for convenience. Therefore, the inter-arrival times are $d_i = t_i - t_{i-1}$ and $d_1 = t_1 - t_0 = \infty$. Assume that D is the modeled cache retention time; that is, D is the maximum duration that a block stays in the LRU cache without any intervening accesses. We can classify the inter-arrival times as follows:

- $d_i \leq D$: A cache hit because the access arrives before the block leaves the cache. The block is moved to the head of the queue because of the LRU eviction.

- $d_i > D$: A cache miss because the access arrives after the block leaves the cache. `AdmitOnMiss` inserts the block into the cache on miss, causing a write to the cache.
- For a flash cache miss, we update the *buffer cache simulator* to see whether it is also a cache miss in the buffer cache. If so, the access is a disk read.

We can then write disk reads $S_b^{\text{AOM}}(D)$, cache byte-time usage² $U_b^{\text{AOM}}(D)$ and bytes written to cache $W_b^{\text{AOM}}(D)$ as functions of D :

$$B_b^{\text{AOM}}(D, i) = \begin{cases} 1, & \text{Buffer Cache Hit at } t_i, \text{ using AOM.} \\ 0, & \text{Buffer Cache Miss at } t_i, \text{ using AOM.} \end{cases}$$

$$S_b^{\text{AOM}}(D) = |\{i : d_i > D, B_b^{\text{AOM}}(D, i) = 0\}|,$$

$$U_b^{\text{AOM}}(D) = \text{Size}(b) \times \sum_i \min(d_i, D),$$

$$W_b^{\text{AOM}}(D) = \text{Size}(b) \times |\{i : d_i > D\}|.$$

Similarly, the metrics for a category C is the sum of the metrics for all blocks in C :

$$S_C^{\text{AOM}}(D) = \sum_{b \in C} S_b^{\text{AOM}}(D),$$

$$U_C^{\text{AOM}}(D) = \sum_{b \in C} U_b^{\text{AOM}}(D),$$

$$W_C^{\text{AOM}}(D) = \sum_{b \in C} W_b^{\text{AOM}}(D).$$

The only difference between `AdmitOnWrite` and `AdmitOnMiss` is that `AdmitOnWrite` also takes into account write accesses. Therefore, for `AdmitOnWrite`, we let t_1 be the write access time, t_2 be the first read access time, t_3 be the second read access time and so on. Then we can similarly define $S_C^{\text{AOW}}(D)$, $U_C^{\text{AOW}}(D)$ and $W_C^{\text{AOW}}(D)$.

For `AdmitOnSecondMiss`, a block is admitted at the second miss (read access). In addition, to prevent the cache from inserting a cold block, we require that when inserting a block, its last read access time must be not older than the oldest last access time of the blocks in the cache. Mathematically, a block is inserted at t_{i-1} (if not already in the cache) only if $d_{i-1} = t_{i-1} - t_{i-2} \leq D$. Therefore, the condition that a block is in the cache at t_{i-1} , either because it is already in the cache or it is inserted, is $d_{i-1} \leq D$, and so an access at t_i is a cache hit if and only if $d_{i-1} \leq D$ and $d_i \leq D$:

$$B_b^{\text{AOSM}}(D, i) = \begin{cases} 1, & \text{Buffer Cache Hit at } t_i, \text{ using AOSM.} \\ 0, & \text{Buffer Cache Miss at } t_i, \text{ using AOSM.} \end{cases}$$

$$S_b^{\text{AOSM}}(D) = |\{i : \max(d_{i-1}, d_i) > D, B_b^{\text{AOSM}}(D, i) = 0\}|.$$

For $U_b^{\text{AOSM}}(D)$, the access at t_i contributes cache usage if either it is a cache hit, $\max(d_i, d_{i-1}) \leq D$, with residence time d_i , or

²Bytes of occupied cache multiplied by seconds of residence time in cache. The same concept is also used in LHD [4].

a block insertion, $d_i \leq D < d_{i-1}$, with residence time D :

$$U_b^{\text{AOSM}}(D) = \text{Size}(b) \times \sum_i \left(d_i \times \mathbf{1}_{\{\max(d_i, d_{i-1}) \leq D\}} + D \times \mathbf{1}_{\{d_i \leq D < d_{i-1}\}} \right).$$

$W_b^{\text{AOSM}}(D)$ is the block size times the number of insertions:

$$W_b^{\text{AOSM}}(D) = \text{Size}(b) \times |\{i : d_i \leq D < d_{i-1}\}|.$$

Of course, $S_C^{\text{AOSM}}(D)$, $U_C^{\text{AOSM}}(D)$ and $W_C^{\text{AOSM}}(D)$ can be defined similarly.

Because `NeverAdmit` does not insert any blocks at all, $U_C^{\text{NA}}(D) = 0$, $W_C^{\text{NA}}(D) = 0$ and $S_C^{\text{NA}}(D)$ is the number of buffer cache misses because of the accesses to the blocks in C :

$$B_b^{\text{NA}}(D, i) = \begin{cases} 1, & \text{Buffer Cache Hit at } t_i, \text{ using NA.} \\ 0, & \text{Buffer Cache Miss at } t_i, \text{ using NA.} \end{cases}$$

$$S_b^{\text{NA}}(D) = |\{i : B_b^{\text{NA}}(D, i) = 0\}|,$$

$$S_C^{\text{NA}}(D) = \sum_{b \in C} S_b^{\text{NA}}(D).$$

A.3 Linear program

We minimize the total cost by formulating a linear program. The cost function is the sum of the cost of disk reads and the cost of written bytes:

$$V_C^p(D) = \text{Cost of } S_C^p(D) + \text{Cost of } W_C^p(D),$$

for $p \in \{\text{AOM}, \text{AOW}, \text{AOSM}, \text{NA}\}$ and a given category C .

A category can receive *fractional* admission policies. For example, `CacheSack` may decide that it is optimal to apply `AdmitOnMiss`, `AdmitOnSecondMiss`, `AdmitOnWrite` and `NeverAdmit` are applied to 30%, 20%, 10% and 40% of blocks in C , respectively. Then we can formulate a linear program that finds optimal policy fractions $\{\alpha_C^{\text{AOM}}, \alpha_C^{\text{AOW}}, \alpha_C^{\text{AOSM}}, \alpha_C^{\text{NA}}\}$ to minimize the overall cost:

$$\min_{\alpha_C^{\text{AOM}}, \alpha_C^{\text{AOW}}, \alpha_C^{\text{AOSM}}, \alpha_C^{\text{NA}}} \sum_C \left(\alpha_C^{\text{AOM}} V_C^{\text{AOM}}(D) + \alpha_C^{\text{AOSM}} V_C^{\text{AOSM}}(D) + \alpha_C^{\text{AOW}} V_C^{\text{AOW}}(D) + \alpha_C^{\text{NA}} V_C^{\text{NA}}(D) \right), \quad (1)$$

subject to the capacity constraint that the cache usage should not exceed the given cache capacity U_{total} :

$$\begin{aligned} 0 \leq \alpha_C^{\text{AOM}}, \alpha_C^{\text{AOW}}, \alpha_C^{\text{AOSM}}, \alpha_C^{\text{NA}} \leq 1, \\ \alpha_C^{\text{AOM}} + \alpha_C^{\text{AOW}} + \alpha_C^{\text{AOSM}} + \alpha_C^{\text{NA}} = 1, \\ \sum_C \left(\alpha_C^{\text{AOM}} U_C^{\text{AOM}}(D) + \alpha_C^{\text{AOSM}} U_C^{\text{AOSM}}(D) + \alpha_C^{\text{AOW}} U_C^{\text{AOW}}(D) + \alpha_C^{\text{NA}} U_C^{\text{NA}}(D) \right) \leq U_{\text{total}}. \end{aligned}$$

We note that if the LRU cache is perfectly modeled by the approach in Section A.2, the resulting cache retention time of the LRU cache is exactly D after applying the optimal policy fractions.

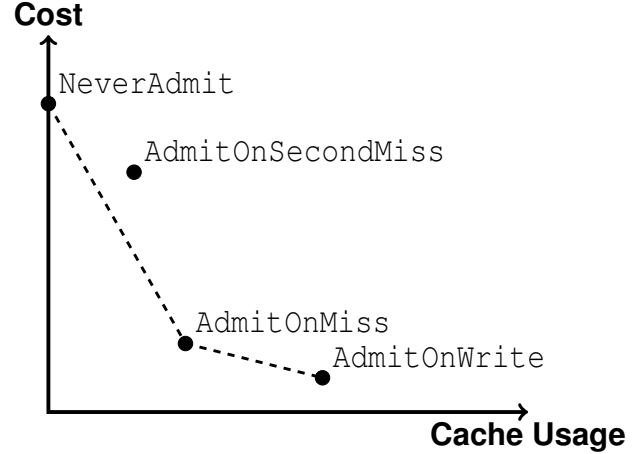


Figure 16: Example of Andrew’s monotone chain convex hull algorithm applied to the admission policies.

A.4 Greedy algorithm

Although the linear program (1) can be solved by a standard solver, we are able to solve it by a greedy algorithm with a simple transformation. It is especially beneficial for the production deployment because of the low-overhead and stability of the greedy algorithm, compared to a generic solver. We first note that the difference between the above linear program and a fractional knapsack problem [14] is that for each category, we need to decide *coupled* four fractions (three degrees of freedom), instead of two fractions (one degree of freedom) in a fractional knapsack problem. Thus, the greedy algorithm in [14] can not be directly applied. However, we can use Andrew’s lower convex hull algorithm [2] to decouple the dependency.

For a given category C , the lower convex hull formed by the points $\{(U_C^p, V_C^p), p \in \{\text{AOM}, \text{AOW}, \text{AOSM}, \text{NA}\}\}$, is the lowest cost of C that can be generated among all convex combinations of the policies. For example, Figure 16 is the lower convex hull constructed by the given admission policies by using Andrew’s algorithm. Let $F_C(u)$ denote the lower convex hull formed above, as a mapping from cache usage u to the corresponding cost, for each category C . By dropping any line segments with non-negative slopes, all F_C are strictly decreasing, piecewise linear functions. Then we can transform the linear program to a convex optimization problem:

$$\min_{u_C \geq 0} \sum_C F_C(u_C), \quad \sum_C u_C \leq U_{\text{total}}.$$

We can then solve the above convex optimization problem by the steepest descent method (a greedy algorithm). We initialize $u_C = 0$ for all C and iteratively decide each u_C in the following way. We first choose the line segment with the most negative slope among all line segments of F_C and change the value of the corresponding u_C . In the same fashion, we

then choose the line segment with second most negative slope and change the value of the corresponding u_C , then the third most negative slope, and so on, until the sum of u_C reaches U_{total} .

Because we allow fractional policies, the category corresponding to the last chosen line segment generally has the optimal policy as a convex combination of two of $\{\text{AOM}, \text{AOW}, \text{AOSM}, \text{NA}\}$, and the optimal policy of any other category must be exactly one of $\{\text{AOM}, \text{AOW}, \text{AOSM}, \text{NA}\}$.

A.5 Optimization over modeled cache retention times

The linear program (1) is to find the optimal policy fractions for a *given* modeled cache retention time D , which can not be known *a priori*. Thus, we need to solve the same optimization problem for *all* possible D :

$$\min_{D>0} \min_{\alpha_C^{\text{AOM}}, \alpha_C^{\text{AOW}}, \alpha_C^{\text{AOSM}}, \alpha_C^{\text{NA}}} \sum_C \left(\alpha_C^{\text{AOM}} v_C^{\text{AOM}}(D) + \alpha_C^{\text{AOSM}} v_C^{\text{AOSM}}(D) + \alpha_C^{\text{AOW}} v_C^{\text{AOW}}(D) + \alpha_C^{\text{NA}} v_C^{\text{NA}}(D) \right),$$

subject to the same capacity constraint.

To do this, we can use a standard scalar-variable optimization approach like Brent’s method [10] for $0 < D \leq \mathbf{D}$, where \mathbf{D} is a suitable upper bound. A brute-force approach may be even more practical for implementation: we simply solve the optimization problem for a set of reasonable retention times: $0 < D_1 < D_2 < \dots < D_m = \mathbf{D}$.

References

- [1] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, François Labelle, Nate Coehlo, Xudong Shi, and C. Eric Schrock. Janus: Optimal Flash Provisioning for Cloud Storage Workloads. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 91–102. USENIX Association, June 2013.
- [2] A.M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.
- [3] Jeff Barr. AQUA (Advanced Query Accelerator) – A Speed Boost for Your Amazon Redshift Queries. <https://aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift/>, April 2021.
- [4] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403. USENIX Association, April 2018.
- [5] Nathan Beckmann and Daniel Sanchez. Maximizing Cache Performance Under Uncertainty. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 109–120, 2017.
- [6] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Syst. J.*, 5(2):78–101, June 1966.
- [7] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosf, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, November 2020.
- [8] Daniel S. Berger. Towards Lightweight and Robust Machine Learning for CDN Caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks, HotNets ’18*, page 134–140. Association for Computing Machinery, 2018.
- [9] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic Caching: Flexible Caching for Web Applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511. USENIX Association, July 2017.
- [10] R.P. Brent. *Algorithms for minimization without derivatives*. Prentice-Hall, 1973.
- [11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.
- [12] F.J. Corbató. *A Paging Experiment with the Multics System*. Massachusetts Institute of Technology, 1968.
- [13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google’s Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264. USENIX Association, October 2012.
- [14] George B. Dantzig. Discrete-Variable Extremum Problems. *Operations Research*, 5(2):266–288, 1957.

- [15] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Trans. Storage*, 13(4), November 2017.
- [16] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78. USENIX Association, February 2019.
- [17] Ronald Fagin. Asymptotic miss ratios over independent references. *Journal of Computer and System Sciences*, 14(2):222–250, 1977.
- [18] Alex Gartrell. McDipper: A key-value cache for Flash storage. <https://engineering.fb.com/2013/03/05/web/mcdipper-a-key-value-cache-for-flash-storage/>, March 2013.
- [19] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, 2003.
- [20] Dean Hildebrand and Denis Serenyi. Colossus under the hood: a peek into Google’s scalable storage system. <https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system>, April 2021.
- [21] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving Flash-Based Disk Cache with Lazy Adaptive Replacement. *ACM Trans. Storage*, 12(2), February 2016.
- [22] Akanksha Jain and Calvin Lin. Back to the Future: Leveraging Belady’s Algorithm for Improved Cache Replacement. In *Proceedings of the 43rd International Symposium on Computer Architecture, ISCA ’16*, page 78–89. IEEE Press, 2016.
- [23] Bo Jiang, Philippe Nain, and Don Towsley. On the Convergence of the TTL Approximation for an LRU Cache under Independent Stationary Request Processes. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(4), September 2018.
- [24] Jinhong Li, Qiuping Wang, Patrick P. C. Lee, and Chao Shi. An In-Depth Analysis of Cloud Block Storage Workloads in Large-Scale Production. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 37–47, 2020.
- [25] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An Imitation Learning Approach for Cache Replacement. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 6237–6247. PMLR, 2020.
- [26] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP ’21*, page 243–262. Association for Computing Machinery, 2021.
- [27] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. USENIX Association, March 2003.
- [28] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP ’11*, page 57–70. Association for Computing Machinery, 2011.
- [29] Changwoo Min, Kangyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies, FAST’12*, page 12. USENIX Association, 2012.
- [30] Pancham Pancham, Deepak Chaudhary, and Ruchin Gupta. Comparison of Cache Page Replacement Techniques to Enhance Cache Memory Performance. *International Journal of Computer Applications*, 98:27–33, 07 2014.
- [31] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’06)*, pages 423–432, 2006.
- [32] Ricardo Santana, Steven Lyons, Ricardo Koller, Raju Rangaswami, and Jason Liu. To ARC or Not to ARC. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*. USENIX Association, July 2015.
- [33] Min Joon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional Attention Flow for Machine Comprehension. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

- [34] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A Distributed SQL Database That Scales. In *VLDB*, 2013.
- [35] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544. USENIX Association, February 2020.
- [36] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced Photo Caching on Flash for Facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386. USENIX Association, February 2015.
- [37] Rajesh Thallam. BigQuery explained: An overview of BigQuery’s architecture. <https://cloud.google.com/blog/products/data-analytics/new-blog-series-bigquery-explained-overview>, September 2020.
- [38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.
- [39] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving Cache Replacement with ML-Based LeCaR. In *Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage’18, page 3. USENIX Association, 2018.
- [40] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache Modeling and Optimization using Miniature Simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498. USENIX Association, July 2017.
- [41] Hua Wang, Xinbo Yi, Ping Huang, Bin Cheng, and Ke Zhou. Efficient SSD Caching by Avoiding Unnecessary Writes Using Machine Learning. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018. Association for Computing Machinery, 2018.
- [42] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, Swaminathan Sundararaman, and Robert Wood. HEC: Improving Endurance of High Performance Flash-Based Cache Devices. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR ’13. Association for Computing Machinery, 2013.
- [43] Giulio Zhou and Martin Maas. Learning on distributed traces for data center storage systems. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 350–364, 2021.
- [44] Huapeng Zhou, Linpeng Tang, Qi Huang, and Wyatt Lloyd. The Evolution of Advanced Caching in the Facebook CDN. <https://research.fb.com/blog/2016/04/the-evolution-of-advanced-caching-in-the-facebook-cdn/>, April 2016.

Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service

*Mostafa Elhemali, Niall Gallagher, Nicholas Gordon, Joseph Idziorek, Richard Krog
Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somu Perianayagam, Tim Rath
Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, Akshat Vig*
dynamodb-paper@amazon.com
Amazon Web Services

Abstract

Amazon DynamoDB is a NoSQL cloud database service that provides consistent performance at any scale. Hundreds of thousands of customers rely on DynamoDB for its fundamental properties: consistent performance, availability, durability, and a fully managed serverless experience. In 2021, during the 66-hour Amazon Prime Day shopping event, Amazon systems - including Alexa, the Amazon.com sites, and Amazon fulfillment centers, made trillions of API calls to DynamoDB, peaking at 89.2 million requests per second, while experiencing high availability with single-digit millisecond performance. Since the launch of DynamoDB in 2012, its design and implementation have evolved in response to our experiences operating it. The system has successfully dealt with issues related to fairness, traffic imbalance across partitions, monitoring, and automated system operations without impacting availability or performance. Reliability is essential, as even the slightest disruption can significantly impact customers. This paper presents our experience operating DynamoDB at a massive scale and how the architecture continues to evolve to meet the ever-increasing demands of customer workloads.

1 Introduction

Amazon DynamoDB is a NoSQL cloud database service that supports fast and predictable performance at any scale. DynamoDB is a foundational AWS service that serves hundreds of thousands of customers using a massive number of servers located in data centers around the world. DynamoDB powers multiple high-traffic Amazon properties and systems including Alexa, the Amazon.com sites, and all Amazon fulfillment centers. Moreover, many AWS services such as AWS Lambda, AWS Lake Formation, and Amazon SageMaker are built on DynamoDB, as well as hundreds of thousands of customer applications.

These applications and services have demanding operational requirements with respect to performance, reliability, durability, efficiency, and scale. The users of DynamoDB rely

on its ability to serve requests with consistent low latency. For DynamoDB customers, consistent performance at any scale is often more important than median request service times because unexpectedly high latency requests can amplify through higher layers of applications that depend on DynamoDB and lead to a bad customer experience. The goal of the design of DynamoDB is to complete *all* requests with low single-digit millisecond latencies. In addition, the large and diverse set of customers who use DynamoDB rely on an ever-expanding feature set as shown in Figure 1. As DynamoDB has evolved over the last ten years, a key challenge has been adding features without impacting operational requirements. To benefit customers and application developers, DynamoDB uniquely integrates the following six fundamental system properties:

DynamoDB is a fully managed cloud service. Using the DynamoDB API, applications create tables and read and write data without regard for where those tables are stored or how they're managed. DynamoDB frees developers from the burden of patching software, managing hardware, configuring a distributed database cluster, and managing ongoing cluster operations. DynamoDB handles resource provisioning, automatically recovers from failures, encrypts data, manages software upgrades, performs backups, and accomplishes other tasks required of a fully-managed service.

DynamoDB employs a multi-tenant architecture. DynamoDB stores data from different customers on the same physical machines to ensure high utilization of resources, enabling us to pass the cost savings to our customers. Resource reservations, tight provisioning, and monitored usage provide isolation between the workloads of co-resident tables.

DynamoDB achieves boundless scale for tables. There are no predefined limits for the amount of data each table can store. Tables grow elastically to meet the demand of the customers' applications. DynamoDB is designed to scale the resources dedicated to a table from several servers to many thousands as needed. DynamoDB spreads an application's data across more servers as the amount of data storage and the demand for throughput requirements grow.

DynamoDB provides predictable performance. The simple

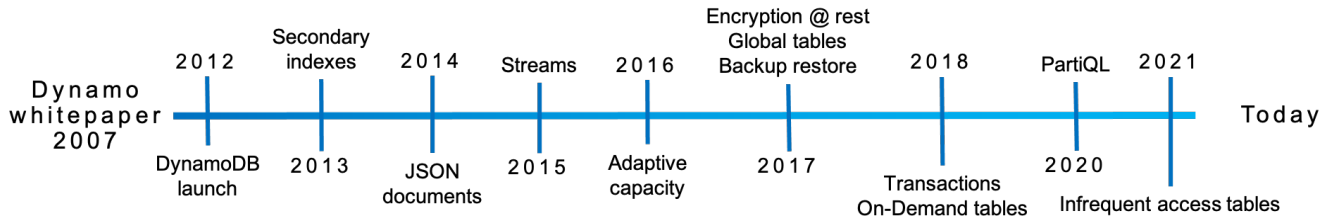


Figure 1: DynamoDB yimeline

DynamoDB API with *GetItem* and *PutItem* operations allows it to respond to requests with consistent low latency. An application running in the same AWS Region as its data will typically see average service-side latencies in the low single-digit millisecond range for a 1 KB item. Most importantly, DynamoDB latencies are predictable. Even as tables grow from a few megabytes to hundreds of terabytes, latencies remain stable due to the distributed nature of data placement and request routing algorithms in DynamoDB. DynamoDB handles any level of traffic through horizontal scaling and automatically partitions and re-partitions data to meet an application’s I/O performance requirements.

DynamoDB is highly available. DynamoDB replicates data across multiple data centers—called Availability Zones in AWS—and automatically re-replicates in the case of disk or node failures to meet stringent availability and durability requirements. Customers can also create global tables that are geo-replicated across selected Regions for disaster recovery and provide low latency access from anywhere. DynamoDB offers an availability SLA of 99.99 for regular tables and 99.999 for global tables (where DynamoDB replicates across tables across multiple AWS Regions).

DynamoDB supports flexible use cases. DynamoDB doesn’t force developers into a particular data model or consistency model. DynamoDB tables don’t have a fixed schema but instead allow each data item to contain any number of attributes with varying types, including multi-valued attributes. Tables use a key-value or document data model. Developers can request strong or eventual consistency when reading items from a table.

In this paper, we describe how DynamoDB evolved as a distributed database service to meet the needs of its customers without losing its key aspect of providing a single-tenant experience to every customer using a multi-tenant architecture. The paper explains the challenges faced by the system and how the service evolved to handle those challenges while connecting the required changes to a common theme of durability, availability, scalability, and predictable performance.

The paper captures the following lessons that we have learnt over the years

- Adapting to customers’ traffic patterns to reshape the physical partitioning scheme of the database tables improves customer experience.

- Performing continuous verification of data-at-rest is a reliable way to protect against both hardware failures and software bugs in order to meet high durability goals.
- Maintaining high availability as a system evolves requires careful operational discipline and tooling. Mechanisms such as formal proofs of complex algorithms, game days (chaos and load tests), upgrade/downgrade tests, and deployment safety provides the freedom to safely adjust and experiment with the code without the fear of compromising correctness.
- Designing systems for predictability over absolute efficiency improves system stability. While components such as caches can improve performance, do not allow them to hide the work that would be performed in their absence, ensuring that the system is always provisioned to handle the unexpected.

The structure of this paper is as follows: Section 2 expands on the history of DynamoDB and explains its origins, which derive from the original Dynamo system. Section 3 presents the architectural overview of DynamoDB. Section 4 covers the journey of DynamoDB from provisioned to on-demand tables. Section 5 covers how DynamoDB ensures strong durability. Section 6 describes the availability challenges faced and how these challenges were handled. Section 7 provides some experimental results based on the Yahoo! Cloud Serving Benchmark (YCSB) benchmarks, and Section 8 concludes the paper.

2 History

The design of DynamoDB was motivated by our experiences with its predecessor Dynamo [9], which was the first NoSQL database system developed at Amazon. Dynamo was created in response to the need for a highly scalable, available, and durable key-value database for shopping cart data. In the early years, Amazon learned that providing applications with direct access to traditional enterprise database instances led to scaling bottlenecks such as connection management, interference between concurrent workloads, and operational problems with tasks such as schema upgrades. Thus, a service-oriented architecture was adopted to encapsulate an application’s data

behind service-level APIs that allowed sufficient decoupling to address tasks like reconfiguration without having to disrupt clients.

High availability is a critical property of a database service as any downtime can impact customers that depend on the data. Another critical requirement for Dynamo was predictable performance so that applications could provide a consistent experience to their users. To achieve these goals, Amazon had to start from first principles when designing Dynamo. The adoption of Dynamo widened to serve several use cases within Amazon because it was the only database service that provided high reliability at scale. However, Dynamo still carried the operational complexity of self-managed large database systems. Dynamo was a single-tenant system and teams were responsible for managing their own Dynamo installations. Teams had to become experts on various parts of the database service and the resulting operational complexity became a barrier to adoption.

During this period, Amazon launched new services (notably Amazon S3 and Amazon SimpleDB) that focused on a managed and elastic experience in order to remove this operational burden. Amazon engineers preferred to use these services instead of managing their own systems like Dynamo, even though the functionality of Dynamo was often better aligned with their applications' needs. Managed elastic services freed developers from administrating databases and allowed them to focus on their applications.

The first database-as-a-service from Amazon was SimpleDB [1], a fully managed elastic NoSQL database service. SimpleDB provided multi-datacenter replication, high availability, and high durability without the need for customers to set up, configure, or patch their database. Like Dynamo, SimpleDB also had a very simple table interface with a restricted query set that served as a building block for many developers. While SimpleDB was successful and powered many applications, it had some limitations. One limitation was that tables had a small capacity in terms of storage (10GB) and of request throughput. Another limitation was the unpredictable query and write latencies, which stemmed from the fact that all table attributes were indexed, and the index needed to be updated with every write. These limitations created a new kind of operational burden for developers. They had to divide data between multiple tables to meet their application's storage and throughput requirements.

We realized that the goal of removing the limitations of SimpleDB and providing a scalable NoSQL database service with predictable performance could not be met with the SimpleDB APIs. We concluded that a better solution would combine the best parts of the original Dynamo design (incremental scalability and predictable high performance) with the best parts of SimpleDB (ease of administration of a cloud service, consistency, and a table-based data model that is richer than a pure key-value store). These architectural discussions culminated in Amazon DynamoDB, a public service launched in 2012

Operation	Description
PutItem	Inserts a new item, or replaces an old item with a new item.
UpdateItem	Updates an existing item, or adds a new item to the table if it doesn't already exist.
DeleteItem	The DeleteItem operation deletes a single item from the table by the primary key.
GetItem	The GetItem operation returns a set of attributes for the item with the given primary key.

Table 1: DynamoDB CRUD APIs for items

that shared most of the name of the previous Dynamo system but little of its architecture. Amazon DynamoDB was the result of everything we'd learned from building large-scale, non-relational databases for Amazon.com and has evolved based on our experiences building highly scalable and reliable cloud computing services at AWS.

3 Architecture

A DynamoDB table is a collection of items, and each item is a collection of attributes. Each item is uniquely identified by a primary key. The schema of the primary key is specified at the table creation time. The primary key schema contains a partition key or a partition and sort key (a composite primary key). The partition key's value is always used as an input to an internal hash function. The output from the hash function and the sort key value (if present) determines where the item will be stored. Multiple items can have the same partition key value in a table with a composite primary key. However, those items must have different sort key values.

DynamoDB also supports secondary indexes to provide enhanced querying capability. A table can have one or more secondary indexes. A secondary index allows querying the data in the table using an alternate key, in addition to queries against the primary key. DynamoDB provides a simple interface to store or retrieve items from a table or an index. Table 1 contains the primary operations available to clients for reading and writing items in DynamoDB tables. Any operation that inserts, updates, or deletes an item can be specified with a condition that must be satisfied in order for the operation to succeed. DynamoDB supports ACID transactions enabling applications to update multiple items while ensuring atomicity, consistency, isolation, and durability (ACID) across items without compromising the scalability, availability, and performance characteristics of DynamoDB tables.

A DynamoDB table is divided into multiple partitions to handle the throughput and storage requirements of the table. Each partition of the table hosts a disjoint and contiguous part of the table's key-range. Each partition has multiple replicas distributed across different Availability Zones for high avail-

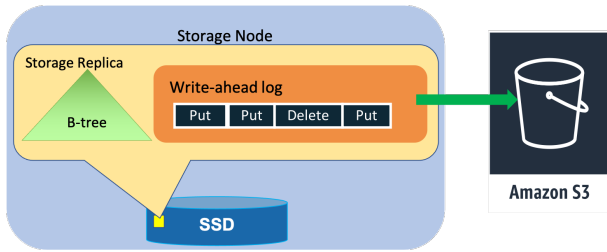


Figure 2: Storage replica on a storage node

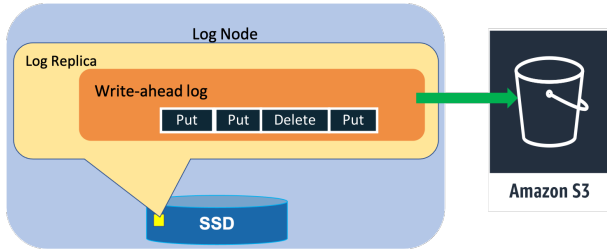


Figure 3: Log replica on a log node

ability and durability. The replicas for a partition form a replication group. The replication group uses Multi-Paxos [14] for leader election and consensus. Any replica can trigger a round of the election. Once elected leader, a replica can maintain leadership as long as it periodically renews its leadership lease.

Only the leader replica can serve write and strongly consistent read requests. Upon receiving a write request, the leader of the replication group for the key being written generates a write-ahead log record and sends it to its peer (replicas). A write is acknowledged to the application once a quorum of peers persists the log record to their local write-ahead logs. DynamoDB supports strongly and eventually consistent reads. Any replica of the replication group can serve eventually consistent reads. The leader of the group extends its leadership using a lease mechanism. If the leader of the group is failure detected (considered unhealthy or unavailable) by any of its peers, the peer can propose a new round of the election to elect itself as the new leader. The new leader won't serve any writes or consistent reads until the previous leader's lease expires.

A replication group consists of storage replicas that contain both the write-ahead logs and the B-tree that stores the key-value data as shown in Figure 2. To improve availability and durability, a replication group can also contain replicas that only persist recent write-ahead log entries as shown in Figure 3. These replicas are called *log replicas*. Log replicas are akin to acceptors in Paxos. Log replicas do not store key-value data. Section 5 and 6 discusses how *log replicas* help DynamoDB improve its availability and durability.

DynamoDB consists of tens of microservices. Some of the core services in DynamoDB are the metadata service, the

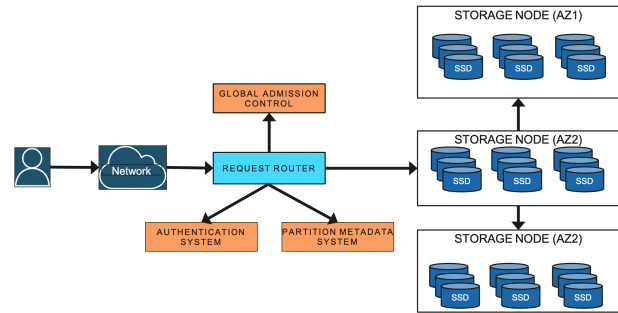


Figure 4: DynamoDB architecture

request routing service, the storage nodes, and the autoadmin service, as shown in Figure 4. The metadata service stores routing information about the tables, indexes, and replication groups for keys for a given table or index. The request routing service is responsible for authorizing, authenticating, and routing each request to the appropriate server. For example, all read and update requests are routed to the storage nodes hosting the customer data. The request routers look up the routing information from the metadata service. All resource creation, update, and data definition requests are routed to the autoadmin service. The storage service is responsible for storing customer data on a fleet of storage nodes. Each of the storage nodes hosts many replicas of different partitions.

The autoadmin service is built to be the central nervous system of DynamoDB. It is responsible for fleet health, partition health, scaling of tables, and execution of all control plane requests. The service continuously monitors the health of all the partitions and replaces any replicas deemed unhealthy (slow or not responsive or being hosted on bad hardware). The service also performs health checks of all core components of DynamoDB and replaces any hardware that is failing or has failed. For example, if the autoadmin service detects a storage node to be unhealthy, it kicks off a recovery process that replaces the replicas hosted on that node to bring the system back to a stable state.

Other DynamoDB services not shown in in Figure 4 support features such as point-in-time restore, on-demand backups, update streams, global admission control, global tables, global secondary indices, and transactions.

4 Journey from provisioned to on-demand

When DynamoDB launched, we introduced an internal abstraction, partitions, as a way to dynamically scale both the capacity and performance of tables. In the original DynamoDB release, customers explicitly specified the throughput that a table required in terms of read capacity units (RCUs) and write capacity units (WCUs). For items up to 4 KB in size, one RCU can perform one strongly consistent read request per second. For items up to 1 KB in size, one WCU can

perform one standard write request per second. RCUs and WCUs collectively are called provisioned throughput. The original system split a table into partitions that allow its contents to be spread across multiple storage nodes and mapped to both the available space and performance on those nodes. As the demands from a table changed (because it grew in size or because the load increased), partitions could be further split and migrated to allow the table to scale elastically. Partition abstraction proved to be really valuable and continues to be central to the design of DynamoDB. However this early version tightly coupled the assignment of both capacity and performance to individual partitions, which led to challenges.

DynamoDB uses admission control to ensure that storage nodes don't become overloaded, to avoid interference between co-resident table partitions, and to enforce the throughput limits requested by customers. Admission control in DynamoDB has evolved over the past decade. Admission control was the shared responsibility of all storage nodes for a table. Storage nodes independently performed admission control based on the allocations of their locally stored partitions. Given that a storage node hosts partitions from multiple tables, the allocated throughput of each partition was used to isolate the workloads. DynamoDB enforced a cap on the maximum throughput that could be allocated to a single partition, and ensured that the total throughput of all the partitions hosted by a storage node is less than or equal to the maximum allowed throughput on the node as determined by the physical characteristics of its storage drives.

The throughput allocated to partitions was adjusted when the overall table's throughput was changed or its partitions were split into child partitions. When a partition was split for size, the allocated throughput of the parent partition was equally divided among the child partitions. When a partition was split for throughput, the new partitions were allocated throughput based on the table's provisioned throughput. For example, assume that a partition can accommodate a maximum provisioned throughput of 1000 WCUs. When a table is created with 3200 WCUs, DynamoDB created four partitions that each would be allocated 800 WCUs. If the table's provisioned throughput was increased to 3600 WCUs, then each partition's capacity would increase to 900 WCUs. If the table's provisioned throughput was increased to 6000 WCUs, then the partitions would be split to create eight child partitions, and each partition would be allocated 750 WCUs. If the table's capacity was decreased to 5000 WCUs, then each partition's capacity would be decreased to 675 WCUs.

The uniform distribution of throughput across partitions is based on the assumptions that an application uniformly accesses keys in a table and the splitting a partition for size equally splits the performance. However, we discovered that application workloads frequently have non-uniform access patterns both over time and over key ranges. When the request rate within a table is non-uniform, splitting a partition and dividing performance allocation proportionately can result

in the *hot* portion of the partition having less available performance than it did before the split. Since throughput was allocated statically and enforced at a partition level, these non-uniform workloads occasionally resulted in an application's reads and writes being rejected, called throttling, even though the total provisioned throughput of the table was sufficient to meet its needs.

Two most commonly faced challenges by the applications were: *hot partitions* and *throughput dilution*. Hot partitions arose in applications that had traffic going consistently towards a few items of their tables. The hot items could belong to a stable set of partitions or could hop around to different partitions over time. Throughput dilution was common for tables where partitions were split for size. Splitting a partition for size would cause the throughput of the partition to be divided equally among the newly created child partitions, and hence the per partition throughput would decrease.

In both cases, from the customer's perspective, throttling caused their application to experience periods of unavailability even though the service was behaving as expected. Customers who experienced throttling would work around it by increasing a table's provisioned throughput and not use all the capacity. That is, tables would be over-provisioned. While this allowed them to achieve the performance they needed, it was a poor experience because it was difficult to estimate the right level of performance provisioning for their tables.

4.1 Initial improvements to admission control

As we mentioned at the start of this section, hot partitions and throughput dilution stemmed from tightly coupling a rigid performance allocation to each partition, and dividing that allocation as partitions split. We liked that enforcing allocations at an individual partition level avoided the need for the complexities of distributed admission control, but it became clear these controls weren't sufficient. Shortly after launching, DynamoDB introduced two improvements, bursting and adaptive capacity, to address these concerns.

4.1.1 Bursting

The key observation that partitions had non-uniform access also led us to observe that not all partitions hosted by a storage node used their allocated throughput simultaneously. Hence, to absorb temporal spikes in workloads at a partition level, DynamoDB introduced the concept of bursting. The idea behind bursting was to let applications tap into the unused capacity at a partition level on a best effort basis to absorb short-lived spikes. DynamoDB retained a portion of a partition's unused capacity for later bursts of throughput usage for up to 300 seconds and utilized it when consumed capacity exceeded the provisioned capacity of the partition. The unused capacity is called *burst* capacity.

DynamoDB still maintained workload isolation by ensuring

that a partition could only burst if there was unused throughput at the node level. The capacity was managed on the storage node using multiple token buckets: two for each partition (allocated and burst) and one for the node. These buckets provided admission control. When a read or write request arrived on a storage node, if there were tokens in the partition's allocated token bucket, then the request was admitted and tokens were deducted from the partition and node level bucket. Once a partition had exhausted all the provisioned tokens, requests were allowed to burst only when tokens were available both in the burst token bucket and the node level token bucket. Read requests were accepted based on the local token buckets. Write requests using burst capacity required an additional check on the node-level token bucket of other member replicas of the partition. The leader replica of the partition periodically collected information about each of the members node-level capacity. In section 4.3 we explain how we increased a node's ability to burst.

4.1.2 Adaptive capacity

DynamoDB launched adaptive capacity to better absorb long-lived spikes that cannot be absorbed by the burst capacity. Adaptive capacity allowed DynamoDB to better absorb workloads that had heavily skewed access patterns across partitions. Adaptive capacity actively monitored the provisioned and consumed capacity of all the tables. If a table experienced throttling and the table level throughput was not exceeded, then it would automatically increase (boost) the allocated throughput of the partitions of the table using a proportional control algorithm. If the table was consuming more than its provisioned capacity then capacity of the partitions which received the boost would be decreased. The autoadmin system ensured that partitions receiving boost were relocated to an appropriate node that had the capacity to serve the increased throughput, however like bursting, adaptive capacity was also best-effort but eliminated over 99.99% of the throttling due to skewed access pattern.

4.2 Global admission control

Even though DynamoDB had substantially reduced the throughput problem for non-uniform access using bursting and adaptive capacity, both the solutions had limitations. Bursting was only helpful for short-lived spikes in traffic and it was dependent on the node having throughput to support bursting. Adaptive capacity was reactive and kicked in only after throttling had been observed. This meant that the application using the table had already experienced brief period of unavailability. The salient takeaway from bursting and adaptive capacity was that we had tightly coupled partition level capacity to admission control. Admission control was distributed and performed at a partition level. DynamoDB realized it would going to be beneficial to remove admission

control from the partition and let the partition burst *always* while providing workload isolation.

To solve the problem of admission control, DynamoDB replaced adaptive capacity with global admission control (GAC). GAC builds on the same idea of token buckets. The GAC service centrally tracks the total consumption of the table capacity in terms of tokens. Each request router maintains a local token bucket to make admission decisions and communicates with GAC to replenish tokens at regular intervals (in the order of few seconds). GAC maintains an ephemeral state computed on the fly from client requests. Each GAC server can be stopped and restarted without any impact on the overall operation of the service. Each GAC server can track one or more token buckets configured independently. All the GAC servers are part of an independent hash ring. Request routers manage several time-limited tokens locally. When a request from the application arrives, the request router deducts tokens. Eventually, the request router will run out of tokens because of consumption or expiry. When the request router runs of tokens, it requests more tokens from GAC. The GAC instance uses the information provided by the client to estimate the global token consumption and vends tokens available for the next time unit to the client's share of overall tokens. Thus, it ensures that non-uniform workloads that send traffic to only a subset of items can execute up to the maximum partition capacity.

In addition to the global admission control scheme, the partition-level token buckets were retained for defense-in-depth. The capacity of these token buckets is then capped to ensure that one application doesn't consume all or a significant share of the resources on the storage nodes.

4.3 Balancing consumed capacity

Letting partitions always burst required DynamoDB to manage burst capacity effectively. DynamoDB runs on a variety of hardware instance types. These instance types vary by throughput and storage capabilities. The latest generation of storage nodes hosts thousands of partition replicas. The partitions hosted on a single storage node could be wholly unrelated and belong to different tables. Hosting replicas from multiple tables on a storage node, where each table could be from a different customer and have varied traffic patterns involves defining an allocation scheme that decides which replicas can safely co-exist without violating critical properties such as availability, predictable performance, security, and elasticity.

Colocation was a straightforward problem with provisioned throughput tables. Colocation was more manageable in the provisioned mode because of static partitions. Static partitions made the allocation scheme reasonably simple. In the case of provisioned tables without bursting and adaptive capacity, allocation involved finding storage nodes that could accommodate a partition based on its allocated capacity. Partitions

were never allowed to take more traffic than their allocated capacity and, hence there were no noisy neighbors. All partitions on a storage node did not utilize their total capacity at a given instance. Bursting when trying to react to the changing workload meant that the storage node might go above its prescribed capacity and thus made the colocation of tenants a more complex challenge. Thus, the system packed storage nodes with a set of replicas greater than the node's overall provisioned capacity. DynamoDB implemented a system to proactively balance the partitions allocated across the storage nodes based on throughput consumption and storage to mitigate availability risks caused by tightly packed replicas. Each storage node independently monitors the overall throughput and data size of all its hosted replicas. In case the throughput is beyond a threshold percentage of the maximum capacity of the node, it reports to the autoadmin service a list of candidate partition replicas to move from the current node. The autoadmin finds a new storage node for the partition in the same or another Availability Zone that doesn't have a replica of this partition.

4.4 Splitting for consumption

Even with GAC and the ability for partitions to burst always, tables could experience throttling if their traffic was skewed to a specific set of items. To address this problem, DynamoDB automatically scales out partitions based on the throughput consumed. Once the consumed throughput of a partition crosses a certain threshold, the partition is split for consumption. The split point in the key range is chosen based on key distribution the partition has observed. The observed key distribution serves as a proxy for the application's access pattern and is more effective than splitting the key range in the middle. Partition splits usually complete in the order of minutes. There are still class of workloads that cannot benefit from split for consumption. For example, a partition receiving high traffic to a single item or a partition where the key range is accessed sequentially will not benefit from split. DynamoDB detects such access patterns and avoids splitting the partition.

4.5 On-demand provisioning

Many applications that migrated to DynamoDB previously ran on-premises or on self-hosted databases. In either scenario, the application developer had to provision servers. DynamoDB provides a simplified serverless operational model and a new model for provisioning - read and write capacity units. Because the concept of capacity units was new to customers, some found it challenging to forecast the provisioned throughput. As mentioned in the beginning of this section, customers either over provisioned, which resulted in low utilization or under provisioned which resulted in throttles. To improve the customer experience for spiky workloads, we

launched on-demand tables. On-demand tables remove the burden from our customers of figuring out the right provisioning for tables. DynamoDB provisions the on-demand tables based on the consumed capacity by collecting the signal of reads and writes and instantly accommodates up to double the previous peak traffic on the table. If an application needs more than double the previous peak on table, DynamoDB automatically allocates more capacity as the traffic volume increases to ensure that the workload does not experience throttling. On-demand scales a table by splitting partitions for consumption. The split decision algorithm is based on traffic. GAC allows DynamoDB to monitor and protect the system from one application consuming all the resources. The ability to balance based on consumed capacity effectively means partitions of on-demand tables can be placed intelligently so as to not run into node level limits.

5 Durability and correctness

Data should never be lost after it has been committed. In practice, data loss can occur because of hardware failures, software bugs, or hardware bugs. DynamoDB is designed for high durability by having mechanisms to prevent, detect, and correct any potential data losses.

5.1 Hardware failures

As with most database management systems, the write-ahead logs [15] in DynamoDB are central for providing durability and crash recovery. Write ahead logs are stored in all three replicas of a partition. For higher durability, the write ahead logs are periodically archived to S3, an object store that is designed for 11 nines of durability. Each replica still contains the most recent write-ahead logs that are usually waiting to be archived. The unarchived logs are typically a few hundred megabytes in size. In a large service, hardware failures such as memory and disk failures are common. When a node fails, all replication groups hosted on the node are down to two copies. The process of healing a storage replica can take several minutes because the repair process involves copying the B-tree and write-ahead logs. Upon detecting an unhealthy storage replica, the leader of a replication group adds a log replica to ensure there is no impact on durability. Adding a log replica takes only a few seconds because the system has to copy only the recent write-ahead logs from a healthy replica to the new replica without the B-tree. Thus, quick healing of impacted replication groups using log replicas ensures high durability of most recent writes.

5.2 Silent data errors

Some hardware failures can cause incorrect data to be stored [5, 7]. In our experience, these errors can happen because of the storage media, CPU, or memory [5]. Unfortunately, it's

very difficult to detect these and they can happen anywhere in the system. DynamoDB makes extensive use of checksums to detect silent errors. By maintaining checksums within every log entry, message, and log file, DynamoDB validates data integrity for every data transfer between two nodes. These checksums serve as guardrails to prevent errors from spreading to the rest of the system. For example, a checksum is computed for every message between nodes or components and is verified because these messages can go through various layers of transformations before they reach their destination. Without such checks, any of the layers could introduce a silent error.

Every log file that is archived to S3 has a manifest that contains information about the log, such as a table, partition and start and end markers for the data stored in the log file. The agent responsible for archiving log files to S3 performs various checks before uploading the data. These include and are not limited to verification of every log entry to ensure that it belongs to the correct table and partition, verification of checksums to detect any silent errors, and verification that the log file doesn't have any holes in the sequence numbers. Once all the checks are passed, the log file and its manifest are archived. Log archival agents run on all three replicas of the replication group. If one of the agents finds that a log file is already archived, the agent downloads the uploaded file to verify the integrity of the data by comparing it with its local write-ahead log. Every log file and manifest file are uploaded to S3 with a content checksum. The content checksum is checked by S3 as part of the put operation, which guards against any errors during data transit to S3.

5.3 Continuous verification

DynamoDB also continuously verifies data at rest. Our goal is to detect any silent data errors or bit rot in the system. An example of such a continuous verification system is the *scrub* process. The goal of scrub is to detect errors that we had not anticipated, such as bit rot. The scrub process runs and verifies two things: all three copies of the replicas in a replication group have the same data, and the data of the live replicas matches with a copy of a replica built offline using the archived write-ahead log entries. The process of building a replica using archived logs is explained in section 5.5 below. The verification is done by computing the checksum of the live replica and matching that with a snapshot of one generated from the log entries archived in S3. The scrub mechanism acts as a defense in depth to detect divergences between the live storage replicas with the replicas built using the history of logs from the inception of the table. These comprehensive checks have been very beneficial in providing confidence in the running system. A similar technique of continuous verification is used to verify replicas of global tables. Over the years, we have learned that continuous verification of data-at-rest is the most reliable method of protecting against hardware

failures, silent data corruption, and even software bugs.

5.4 Software bugs

DynamoDB is a distributed key-value store that's built on a complex substrate. High complexity increases the probability of human error in design, code, and operations. Errors in the system could cause loss or corruption of data, or violate other interface contracts that our customers depend on. We use formal methods [16] extensively to ensure the correctness of our replication protocols. The core replication protocol was specified using TLA+ [12, 13]. When new features that affect the replication protocol are added, they are incorporated into the specification and model checked. Model checking has allowed us to catch subtle bugs that could have led to durability and correctness issues before the code went into production. Other services such as S3 [6] have also found model-checking useful in similar scenarios.

We also employ extensive failure injection testing and stress testing to ensure the correctness of every piece of software deployed. In addition to testing and verifying the replication protocol of the data plane, formal methods have also been used to verify the correctness of our control plane and features such as distributed transactions.

5.5 Backups and restores

In addition to guarding against physical media corruption, DynamoDB also supports backup and restore to protect against any logical corruption due to a bug in a customer's application. Backups or restores don't affect performance or availability of the table as they are built using the write-ahead logs that are archived in S3. The backups are consistent across multiple partitions up to the nearest second. The backups are full copies of DynamoDB tables and are stored in an Amazon S3 bucket. Data from a backup can be restored to a new DynamoDB table at any time.

DynamoDB also supports point-in-time restore. Using point-in-time restore, customers can restore the contents of a table that existed at any time in the previous 35 days to a different DynamoDB table in the same region. For tables with the point-in-time restore enabled, DynamoDB creates periodic snapshots of the partitions that belong to the table and uploads them to S3. The periodicity at which a partition is snapshotted is decided based on the amount of write-ahead logs accumulated for the partition. The snapshots, in conjunction to write-ahead logs, are used to do point-in-time restore. When a point-in-time restore is requested for a table, DynamoDB identifies the closest snapshots to the requested time for all the partitions of the tables, applies the logs up to the timestamp in the restore request, creates a snapshot of the table, and restores it.

6 Availability

To achieve high availability, DynamoDB tables are distributed and replicated across multiple Availability Zones (AZ) in a Region. DynamoDB regularly tests resilience to node, rack, and AZ failures. For example, to test the availability and durability of the overall service, power-off tests are exercised. Using realistic simulated traffic, random nodes are powered off using a job scheduler. At the end of all the power-off tests, the test tools verify that the data stored in the database is logically valid and not corrupted. This section expands on some of the challenges solved in the last decade to ensure high availability.

6.1 Write and consistent read availability

A partition's write availability depends on its ability to have a healthy leader and a healthy write quorum. A healthy write quorum in the case of DynamoDB consists of two out of the three replicas from different AZs. A partition remains available as long as there are enough healthy replicas for a write quorum and a leader. A partition will become unavailable for writes if the number of replicas needed to achieve the minimum quorum are unavailable. If one of the replicas is unresponsive, the leader adds a log replica to the group. Adding a log replica is the fastest way to ensure that the write quorum of the group is always met. This minimizes disruption to write availability due to an unhealthy write quorum. The leader replica serves consistent reads. Introducing log replicas was a big change to the system, and the formally proven implementation of Paxos provided us the confidence to safely tweak and experiment with the system to achieve higher availability. We have been able to run millions of Paxos groups in a Region with log replicas. Eventually consistent reads can be served by any of the replicas. In case a leader replica fails, other replicas detect its failure and elect a new leader to minimize disruptions to the availability of consistent reads.

6.2 Failure detection

A newly elected leader will have to wait for the expiry of the old leader's lease before serving any traffic. While this only takes a couple of seconds, the elected leader cannot accept any new writes or consistent read traffic during that period, thus disrupting availability. One of the critical components for a highly available system is failure detection for the leader. Failure detection must be quick and robust to minimize disruptions. False positives in failure detection can lead to more disruptions in availability. Failure detection works well for failure scenarios where every replica of the group loses connection to the leader. However, nodes can experience gray network failures. Gray network failures can happen because of communication issues between a leader and follower, is-

ues with outbound or inbound communication of a node, or front-end routers facing communication issues with the leader even though the leader and followers can communicate with each other. Gray failures can disrupt availability because there might be a false positive in failure detection or no failure detection. For example, a replica that isn't receiving heartbeats from a leader will try to elect a new leader. As mentioned in the section above, this can disrupt availability. To solve the availability problem caused by gray failures, a follower that wants to trigger a failover sends a message to other replicas in the replication group asking if they can communicate with the leader. If replicas respond with a healthy leader message, the follower drops its attempt to trigger a leader election. This change in the failure detection algorithm used by DynamoDB significantly minimized the number of false positives in the system, and hence the number of spurious leader elections.

6.3 Measuring availability

DynamoDB is designed for 99.999 percent availability for global tables and 99.99 percent availability for Regional tables. Availability is calculated for each 5-minute interval as the percentage of requests processed by DynamoDB that succeed. To ensure these goals are being met, DynamoDB continuously monitors availability at service and table levels. The tracked availability data is used to analyze customer perceived availability trends and trigger alarms if customers see errors above a certain threshold. These alarms are called customer-facing alarms (CFA). The goal of these alarms is to report any availability-related problems and proactively mitigate the problem either automatically or through operator intervention. In addition to real-time tracking, the system runs daily jobs that trigger aggregation to calculate aggregate availability metrics per customer. The results of the aggregation are uploaded to S3 for regular analysis of availability trends.

DynamoDB also measures and alarms on availability observed on the client-side. There are two sets of clients used to measure the user-perceived availability. The first set of clients are internal Amazon services using DynamoDB as the data store. These services share the availability metrics for DynamoDB API calls as observed by their software. The second set of clients is our DynamoDB canary applications. These applications are run from every AZ in the Region, and they talk to DynamoDB through every public endpoint. Real application traffic allows us to reason about DynamoDB availability and latencies as seen by our customers and catch gray failures [10, 11]. They are a good representation of what our customers might be experiencing both as long and short-term trends.

6.4 Deployments

Unlike a traditional relational database, DynamoDB takes care of deployments without the need for maintenance win-

dows and without impacting the performance and availability that customers experience. Software deployments are done for various reasons, including new features, bug fixes, and performance optimizations. Often deployments involve updating numerous services. DynamoDB pushes software updates at a regular cadence. A deployment takes the software from one state to another state. The new software being deployed goes through a full development and test cycle to build confidence in the correctness of the code. Over the years, across multiple deployments, DynamoDB has learned that it's not just the end state and the start state that matter; there could be times when the newly deployed software doesn't work and needs a rollback. The rolled-back state might be different from the initial state of the software. The rollback procedure is often missed in testing and can lead to customer impact. DynamoDB runs a suite of upgrade and downgrade tests at a component level before every deployment. Then, the software is rolled back on purpose and tested by running functional tests. DynamoDB has found this process valuable for catching issues that otherwise would make it hard to rollback if needed.

Deploying software on a single node is quite different from deploying software to multiple nodes. The deployments are not atomic in a distributed system, and, at any given time, there will be software running the old code on some nodes and new code on other parts of the fleet. The additional challenge with distributed deployments is that the new software might introduce a new type of message or change the protocol in a way that old software in the system doesn't understand. DynamoDB handles these kinds of changes with read-write deployments. Read-write deployment is completed as a multi-step process. The first step is to deploy the software to read the new message format or protocol. Once all the nodes can handle the new message, the software is updated to send new messages. New messages are enabled with software deployment as well. Read-write deployments ensure that both types of messages can coexist in the system. Even in the case of rollbacks, the system can understand both old and new messages.

All the deployments are done on a small set of nodes before pushing them to the entire fleet of nodes. The strategy reduces the potential impact of faulty deployments. DynamoDB sets alarm thresholds on availability metrics (mentioned in section 6.3). If error rates or latency exceed the threshold values during deployments, the system triggers automatic rollbacks. Software deployments to storage nodes trigger leader failovers that are designed to avoid any impact to availability. The leader replicas relinquish leadership and hence the group's new leader doesn't have to wait for the old leader's lease to expire.

6.5 Dependencies on external services

To ensure high availability, all the services that DynamoDB depends on in the request path should be more highly available than DynamoDB. Alternatively, DynamoDB should be able to continue to operate even when the services on which it depends are impaired. Examples of services DynamoDB depends on for the request path include AWS Identity and Access Management Services (IAM) [2], and AWS Key Management Service (AWS KMS) [3] for tables encrypted using customer keys. DynamoDB uses IAM and AWS KMS to authenticate every customer request. While these services are highly available, DynamoDB is designed to operate when these services are unavailable without sacrificing any of the security properties that these systems provide.

In the case of IAM and AWS KMS, DynamoDB employs a statically stable design [18], where the overall system keeps working even when a dependency becomes impaired. Perhaps the system doesn't see any updated information that its dependency was supposed to have delivered. However, everything before the dependency became impaired continues to work despite the impaired dependency. DynamoDB caches result from IAM and AWS KMS in the request routers that perform the authentication of every request. DynamoDB periodically refreshes the cached results asynchronously. If IAM or KMS were to become unavailable, the routers will continue to use the cached results for pre-determined extended period. Clients that send operations to request routers that don't have the cached results will see an impact. However, we have seen a minimal impact in practice when AWS KMS or IAM is impaired. Moreover, caches improve response times by removing the need to do an off-box call, which is especially valuable when the system is under high load.

6.6 Metadata availability

One of the most important pieces of metadata the request routers needs is the mapping between a table's primary keys and storage nodes. At launch, DynamoDB stored the metadata in DynamoDB itself. This routing information consisted of all the partitions for a table, the key range of each partition, and the storage nodes hosting the partition. When a router received a request for a table it had not seen before, it downloaded the routing information for the entire table and cached it locally. Since the configuration information about partition replicas rarely changes, the cache hit rate was approximately 99.75 percent. The downside is that caching introduces bimodal behavior. In the case of a cold start where request routers have empty caches, every DynamoDB request would result in a metadata lookup, and so the service had to scale to serve requests at the same rate as DynamoDB. This effect has been observed in practice when new capacity is added to the request router fleet. Occasionally the metadata service traffic would spike up to 75 percent. Thus, introducing new

request routers impacted the performance and could make the system unstable. In addition, an ineffective cache can cause cascading failures to other parts of the system as the source of data falls over from too much direct load [4].

DynamoDB wanted to remove and significantly reduce the reliance on the local cache for request routers and other metadata clients without impacting the latency of the customer requests. When servicing a request, the router needs only information about the partition hosting the key for the request. Therefore, it was wasteful to get the routing information for the entire table, especially for large tables with many partitions. To mitigate against metadata scaling and availability risks in a cost-effective fashion, DynamoDB built an in-memory distributed datastore called *MemDS*. MemDS stores all the metadata in memory and replicates it across the MemDS fleet. MemDS scales horizontally to handle the entire incoming request rate of DynamoDB. The data is highly compressed. The MemDS process on a node encapsulates a Perkle data structure, a hybrid of a Patricia tree [17] and a Merkle tree. The Perkle tree allows keys and associated values to be inserted for subsequent lookup using the full key or a key prefix. Additionally, as keys are stored in sorted order, range queries such as *lessThan*, *greaterThan*, and *between* are also supported. The MemDS Perkle tree additionally supports two special lookup operations: *floor* and *ceiling*. The *floor* operation accepts a key and returns a stored entry from the Perkle whose key is less than or equal to the given key. The *ceiling* operation is similar but returns the entry whose key is greater than or equal to the given key.

A new partition map cache was deployed on each request router host to avoid the bi-modality of the original request router caches. In the new cache, a cache hit also results in an asynchronous call to MemDS to refresh the cache. Thus, the new cache ensures the MemDS fleet is always serving a constant volume of traffic regardless of cache hit ratio. The constant traffic to the MemDS fleet increases the load on the metadata fleet compared to the conventional caches where the traffic to the backend is determined by cache hit ratio, but prevents cascading failures to other parts of the system when the caches become ineffective.

DynamoDB storage nodes are the authoritative source of partition membership data. Partition membership updates are pushed from storage nodes to MemDS. Each partition membership update is propagated to all MemDS nodes. If the partition membership provided by MemDS is stale, then the incorrectly contacted storage node either responds with the latest membership if known or responds with an error code that triggers another MemDS lookup by the request router.

7 Micro benchmarks

To show that scale doesn't affect the latencies observed by applications, we ran YCSB [8] workloads of types A (50 percent reads and 50 percent updates) and B (95 percent reads

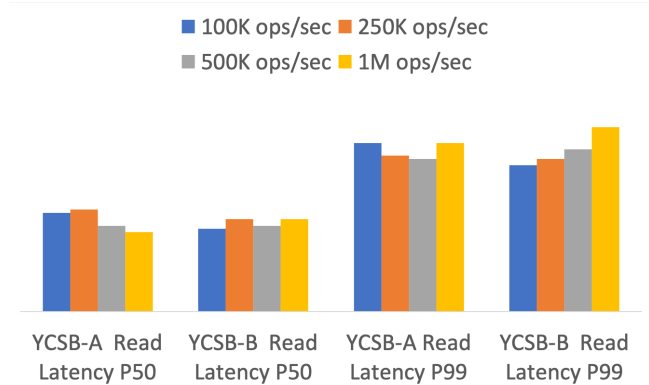


Figure 5: Summary of YCSB read latencies

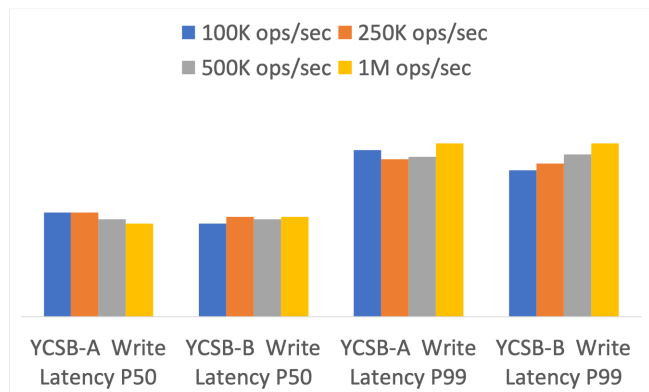


Figure 6: Summary of YCSB write latencies

and 5 percent updates). Both benchmarks used a uniform key distribution and items of size 900 bytes. The workloads were run against production DynamoDB in the North Virginia region. The workloads were scaled from 100 thousand total operations per second to 1 million total operations per second. Figure 5 shows the read latencies of both workloads at 50th and 99th percentiles. The purpose of the graph is to show, even at different throughput, DynamoDB read latencies show very little variance and remain identical even as the throughput of the workload is increased. The read throughput of the Workload B is twice that Workload A and still the latencies show very little variance. Figure 6 shows the writes latencies of both workloads at 50th and 99th percentiles. Like the read latencies, the write latencies remain constant no matter the throughput of the workload. In case of YCSB, workload A drives a higher throughput than workload B, but the write latency profile for both workloads are similar.

8 Conclusion

DynamoDB has pioneered the space of cloud-native NoSQL databases. It is a critical component of thousands of applica-

tions used daily for shopping, food, transportation, banking, entertainment, and so much more. Developers rely on its ability to scale data workloads while providing steady performance, high availability, and low operational complexity. For more than 10 years, DynamoDB has maintained these key properties and extended its appeal to application developers with game-changing features such as on-demand capacity, point-in-time backup and restore, multi-Region replication, and atomic transactions.

9 Acknowledgements

DynamoDB has benefited greatly from its customers whose continuous feedback pushed us to innovate on their behalf. We have been lucky to have an amazing team working with us on this journey. We thank Shawn Bice, Rande Blackman, Marc Brooker, Lewis Bruck, Andrew Certain, Raju Gulabani, James Hamilton, Long Huang, Yossi Levanoni, David Lutz, Maximiliano Maccanti, Rama Pokkunuri, Tony Petrossian, Jim Scharf, Khawaja Shams, Stefano Stefani, Allan Vermuellen, Wei Xiao and the entire DynamoDB team for the impactful contributions during the course of this evolution. Many people have helped to improve this paper. We thank the anonymous reviewers who helped shape the paper. Special thanks to Darcy Jayne, Kiran Reddy, and Andy Warfield for going above and beyond to help.

References

- [1] Amazon SimpleDB: Simple Database Service. <https://aws.amazon.com/simpledb/>.
- [2] AWS Identity and Account Management Service. <https://aws.amazon.com/iam/>.
- [3] AWS Key Management Service. <https://aws.amazon.com/kms/>.
- [4] Summary of the amazon dynamodb service disruption and related impacts in the us-east region. 2015. <https://aws.amazon.com/message/5467D2/>.
- [5] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder. An analysis of data corruption in the storage stack. *ACM Transactions on Storage (TOS)*, 4(3):1–28, 2008.
- [6] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. Van Geffen, and A. Warfield. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21*, page 836–850, New York, NY, USA, 2021. Association for Computing Machinery.
- [7] C. Constantinescu, I. Parulkar, R. Harper, and S. Michalak. Silent data corruption—myth or reality? In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 108–109. IEEE, 2008.
- [8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.
- [9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Voshall, and W. Vogels. Dynamo: Amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, oct 2007.
- [10] T. Hauer, P. Hoffmann, J. Lunney, D. Ardelean, and A. Diwan. Meaningful availability. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 545–557, 2020.
- [11] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The achilles’ heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 150–155, 2017.
- [12] L. Lamport. *Specifying systems*, volume 388. Addison-Wesley Boston, 2002.
- [13] L. Lamport. The pluscal algorithm language. In *International Colloquium on Theoretical Aspects of Computing*, pages 36–60. Springer, 2009.
- [14] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
- [15] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.
- [16] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.
- [17] K. Sklower. A tree-based packet routing table for berkeley unix. In *USENIX Winter*, volume 1991, pages 93–99. Citeseer, 1991.
- [18] B. Weiss and M. Furr. Static stability using availability zones. <https://aws.amazon.com/builders-library/static-stability-using-availability-zones/>.